Practical Mechanisms for Reducing Processor–Memory Data Movement in Modern Workloads

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Amirali Boroumand

M.S., Electrical & Computer Engineering, Carnegie Mellon University
B.S., Computer Engineering, Sharif University of Technology

Carnegie Mellon University
Pittsburgh, PA
December, 2020

Acknowledgments

My Ph.D. journey has been an exciting period full of challenges and learning opportunities, on both a personal and academic level. I am grateful to everyone who helped me to pursue this long and intense journey. First and foremost, I would like to thank my advisors, Prof. Onur Mutlu and Dr. Saugata Ghose. Onur gave me the opportunity to join SAFARI and provided me with tremendous guidance, resources and mentorship throughout my Ph.D. journey. Working with Onur played a major role in my personal growth and realizing my true potential. Onur taught me to always strive higher, an extremely valuable lesson, which had a profound impact on my self development and has become the motto of my life. His high standard for clarity in writing and presentation and his relentless passion for conducting top-notch research had an immense effect on my perspective toward academic research and helped me to grow as an independent researcher and become who I am today. I am forever grateful to Onur for fully supporting me through tough times over the last few years and providing me every opportunity to continue my research despite several ups and downs in my personal life. Due to the political climate in the U.S., I was facing many challenge throughout my Ph.D., including not being able to visit my family for 6 years, not being allowed to work for several tech companies, and not being eligible to apply for various academic fellowships. Without Onur's full support, protection and encouragement during those years, I would not have been able to continue my Ph.D. and complete my dissertation. I am also very thankful to him for generously providing me with the valuable resources as well as freedom to pursue my own research ideas. Onur put significant trust in me and provided me with several collaboration opportunities and internships which all played a major role in my research.

I am very grateful to Saugata for not only being a great advisor, but also a great mentor, a true friend and a big brother. I am thankful for his tremendous support and guidance over every step of my Ph.D. journey. Saugata was always open to brainstorm about my ideas and provide valuable and constructive feedback. He taught me a great deal about writing academic manuscripts and how to construct my ideas in a paper. He patiently corrected my silly writing mistakes and helped me to improve my writing skills in the most constructive and encouraging way. I cannot thank him enough for his endless support during my difficult times. He always goes out of his way to make time to listen to my never-ending complains, lift my spirits and push me to grind through those tough periods. I was incredibly fortunate to have him as my co-advisor during the last few years. Without his endless positivity and unwavering support in my times of turmoil, I would certainly have not finished this journey and completed my dissertation.

I am grateful to the members of my Ph.D. committee: Dr. Parthasarathy Ranganathan and Prof. James Hoe for serving on my defense. I am very thankful for their time and valuable feedback and comments on my research, and for making the final steps towards my PhD very smooth. I would like to especially thank Dr. Ranganathan for being a great mentor for me during my Google internships, providing valuable feedback on my research, and always being supportive throughout this journey.

I am grateful to SAFARI group members that were always supportive and gave feedback on my research. Without support from SAFARI members, this dissertation could not have been completed.

I am grateful to Prof. Brandon Lucia for his valuable feedback and collaboration for my PIM coherence project. I am very thankful to my internship mentors and industrial collaborators for making my work in their companies mutually successful for both sides. Thanks to Eric Shiu, Allan Knies, Berkin Akin, Ravi Narayanaswami, and Xiaoyu Ma for being great and highly supportive mentors during my internships at Google. Thanks to Krishna Teja Malladi and Hongzhong Zheng for giving me the freedom to pursue my own ideas and providing me with valuable feedback during my internship at Samsung.

I would like to thank the generous support of our industrial partners. This research was supported in part by our industrial partners Google, Facebook, Microsoft, Intel, VMware, Alibaba, and Huawei. This research was also partially supported by the Semiconductor Research Corporation. I am also grateful to Carnegie Mellon University for supporting me with the Carnegie Institute of Technology Dean's Tuition and Nick G. Vlahakis Graduate Fellowships.

I would like to acknowledge the ultimate support from my family and friends. I am very thankful to my aunt, Mojdeh, who was always there for me from the very first day I came here to the U.S. and treated me as her own child. I am also very thankful to my grandmother and my other aunt, Nazanin, who always supported me with their love and encouragemet. I am deeply grateful to Rojin Bahrami, who has been my biggest emotional support in the U.S. over the last few years. From putting up with my hectic schedule to listening to my long rants and complaints about my life hurdles, she has always been a rock solid supporter for me throughout my PhD journey. Her love and support was immense during my times of turmoil and I cannot thank her enough for always being there for me.

Lastly, I would like to thank my parents for their endless love, encouragement and support. The Ph.D was a very long and intense journey and I could not have done it without their ultimate support and love. My mom, Shahrzad, is the reason behind all my achievements in life and I owe everything I achieved so far to her. She is truly the embodiment of love, strength, and unconditional support. My dad, Masoud, has been a pillar of support for me over these years and made many sacrifices to ensure I can pursue my academic dreams. I will forever be indebted to both of them for the sacrifices they made during my entire academic journey. They were not able to visit their only child for 6 years (due to the travel ban) and I can only imagine how difficult those 6 years must have been for them. Words are not enough to express how grateful I am for them. This thesis is dedicated to my loving parents, Shahrzad and Masoud.

Abstract

Data movement between the memory system and computation units is one of the most critical challenges in designing high performance and energy-efficient computing systems. The high cost of data movement is forcing architects to rethink the fundamental design of computer systems. Recent advances in memory design enable the opportunity for architects to avoid unnecessary data movement by performing *processing-in-memory* (PIM), also known as *near-data processing* (NDP). While PIM can allow many data-intensive applications to avoid moving data from memory to the CPU, it introduces new challenges for system architects and programmers. Our goal in this thesis is to make PIM effective and practical in conventional computing systems. Toward this end, this thesis presents three major directions: (1) examining the suitability of PIM across key workloads, (2) addressing major system challenges for adopting PIM in computing systems, and (3) redesigning applications aware of PIM capability. In line with these three major directions, we propose a series of practical mechanisms to reduce processor–memory data movement in modern workloads:

First, we comprehensively analyze the energy and performance impact of data movement for several widely-used Google consumer workloads. We find that PIM can significantly reduce data movement for all of these workloads, by performing part of the computation close to memory. Each workload contains simple primitives and functions that contribute to a significant amount of the overall data movement. We investigate whether these primitives and functions are feasible to implement using PIM, given the limited area and power constraints of consumer devices. Our analysis shows that offloading these primitives to PIM logic, consisting of either simple cores or specialized accelerators, eliminates a large amount of data movement, and significantly reduces total system energy execution time.

Second, we address one of the key system challenges for communication with PIM logic by proposing an efficient cache coherence support for *near-data accelerators* (NDAs). We find that enforcing coherence with the rest of the system, which is already a major challenge for on-chip accelerators, becomes more difficult for NDAs. This is because (1) the cost of communication between NDAs and CPUs is high, and (2) NDA applications generate a large amount of off-chip data movement. As a result, as we show in this work, existing coherence mechanisms eliminate most of the benefits of NDAs. Based on our observations, we propose 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 exploits this information to avoid performing *unnecessary* coherence requests, and thus, significantly reduces data movement for coherence. We show that CoNDA significantly improves performance and reduces energy consumption compared to prior coherence mechanisms.

Third, we propose a hardware–software co-design approach aware of PIM for edge machine learning (ML) accelerators to enable energy-efficient and high-performance inference execution. We analyze a commercial Edge TPU (tensor processing unit) using 24 Google edge neural network (NN) models (including CNNs, LSTMs, transducers, and RCNNs), and find that the accelerator suffers from three shortcomings, in terms of computational throughput, energy efficiency, and memory access handling. We comprehensively study the characteristics of each NN layer in all of the Google edge models, and find that these shortcomings arise from the one-size-fits-all approach of the accelerator, as there is a high amount of heterogeneity in key layer characteristics both across different models and across different layers in the same model. To combat this inefficiency, we propose a new acceleration framework called Mensa. Mensa incorporates multiple heterogeneous ML edge accelerators (including both on-chip and near-data accelerators), each of which caters to the characteristics of a particular subset of models. At runtime, Mensa schedules each layer to run on the best-suited accelerator, accounting for both efficiency and inter-layer dependencies. We show that Mensa significantly improves inference energy and throughput, while reducing hardware cost and improving area efficiency over the Edge TPU and Eyeriss v2, two state-of-the-art edge ML accelerators.

Lastly, we propose to redesign emerging modern hybrid databases to be aware of PIM capability, to enable real-time analysis. Hybrid transactional and analytical processing (HTAP) database systems can support real-time data analysis without the high costs of synchronizing across separate single-purpose databases. Unfortunately, for many applications that perform a high rate of data updates, state-of-the-art HTAP systems incur significant drops in transactional and/or analytical throughput compared to performing only transactions or only analytics in isolation, due to (1) data movement between the CPU and memory, (2) data update propagation, and (3) consistency costs. We propose Polynesia, a hardware–software co-designed system for in-memory HTAP databases. Polynesia (1) divides the HTAP system into transactional and analytical processing islands, (2) implements custom algorithms and hardware to reduce the costs of update propagation and consistency, and (3) exploits processing-in-memory for the analytical islands to alleviate data movement. We show that Polynesia significantly outperforms three state-of-the-art HTAP systems and reduces energy consumption.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Problem and Thesis Statement

An exponential growth in data volume, combined with increasing demand for data analysis, has resulted in the emergence of a wide range of data-intensive application. These modern and emerging applications must now process very large datasets [14, 95, 101, 102, 195, 210] and have become increasingly ubiquitous in recent years. For example, an object classification algorithm in an augmented reality application typically trains on millions of example images and video clips and performs classification on real-time high-definition video streams [95, 166]. In order to process meaningful information from the large amounts of data, applications turn to artificial intelligence (AI), or machine learning, and data analytics to methodically mine through the data and extract key properties about the dataset. For instance, Internet-of-Things (IoT) applications ingest a large volume of data from various sensors and typically have some form of learning model (e.g., SQL analytics, machine learning, graph analytics) that is applied to the ingested data to make in-the-field decisions (e.g., navigation for self-driving cars, patient monitoring and response in healthcare) [7, 29, 30, 33].

Due to the increasing reliance on manipulating and mining through large sets of data, these modern applications greatly overwhelm the data storage and movement resources of a modern computer. In a contemporary computer, the main memory is not capable of performing any operations on data. As a result, to perform any operation on data that is stored in memory, the data needs to be moved from the memory to the CPU via the memory channel, a pin-limited off-chip bus (e.g., conventional double data rate, or DDR, memories use a 64-bit memory channel [185,187]). To move the data, the CPU must issue a request to the memory controller, which then issues commands across the memory channel to the DRAM module containing the data. The DRAM module then reads and returns the data across the memory channel, and the data moves through the cache hierarchy before being stored in a CPU cache. The CPU can operate on the data only after the data is loaded from the cache into a CPU register.

Unfortunately, for these modern and emerging applications, data movement between the memory system and computation units has become one of the most critical challenges in designing high-performance and energy-efficient computing systems. The data movement bottleneck incurs a heavy penalty in terms of both performance and energy consumption [42,202]. For example, our comprehensive analysis of several popular Google consumer workloads shows that, among the many sources of energy consumption in consumer devices (e.g., CPUs, GPU, special purpose accelerators, memory), data movement accounts for 62.7% of the total system energy. The high cost of data movement is because of three major reasons. First, there is a long latency and significant energy involved in bringing data from DRAM to computation units. Second, it is difficult to send a large number of requests to memory in parallel, in part because of the narrow width of the memory channel. Third, despite the costs of bringing data into memory, much of this data is not reused by the CPU, rendering the caching either highly inefficient or completely unnecessary [14, 15], especially for modern workloads with very large datasets and random access patterns. Today, the total cost of computation, in terms of performance and in terms of energy, is dominated by the cost of data movement for modern data-intensive workloads such as machine learning and data analytics [14,77,97,99,202,275,392].

The high cost of data movement is forcing architects to rethink the fundamental design of computer systems. One potential way to mitigate data movement cost is to execute the data-movement-heavy portions of our applications close to the data. The idea of performing computation closer to the data has been proposed for at least four decades [76, 81, 109, 199, 218, 254, 267, 287,293,333,347], but earlier efforts were not widely adopted due to the difficulty of integrating processing elements for computation with DRAM. Fortunately, recent advances in 3D-stacked memory technology have enabled cost-effective solutions to realize this idea [174, 186, 233, 250]. 3D-stacked DRAM architectures include a dedicated logic layer, that is capable of providing logic functionality, with high-bandwidth low-latency connectivity to DRAM layers. Recent works take advantage of the logic layer to perform *processing-in-memory* (PIM), also known as *near-data processing* (NDP) [14, 15, 16, 44, 77, 97, 99, 139, 144, 167, 168, 206, 210, 260, 261, 275, 294, 356, 380, 381, 389]. PIM allows the CPU to dispatch parts of the application for execution on compute units that are close to DRAM. Offloading computation using PIM has two major benefits. First, it eliminates a significant portion of the data movement between main memory and conventional processors. Second, it can take advantage of the high-bandwidth and low-latency access to the data inside 3D-stacked DRAM.

While PIM can allow many data-intensive applications to avoid moving data from memory to the CPU, it introduces new challenges for system architects and programmers. First, programmers need to identify primitives that can benefit from PIM and can be implemented given the area and power constraints of 3D-stacked memory's logic layer. Whether to execute part or all of an application in memory depends on: (1) architectural constraints, such as area and energy limitations, and the type of logic implementable within memory; and (2) application properties, such as the intensities of computation and memory accesses, and the amount of data shared across different functions. Second, system architects and programmers must establish efficient interfaces and mechanisms that allow programs to easily take advantage of the benefits of PIM. In particular, the processing logic inside memory does not have quick access to important mechanisms required by modern programs and systems, such as cache coherence, which programmers rely on for software development productivity. Finally, to fully benefit from PIM, a software–hardware co-design approach is required to redesign applications aware of PIM capability. In other words, applications can fully benefit from PIM if both the hardware and software are co-designed to take advantage of PIM.

Our goal in this thesis is to make PIM effective and practical in conventional computing systems. Our approach can be summarized in the following thesis statement:

Processor–memory data movement can be significantly reduced using practical mechanisms *that are aware of modern workloads and architectural constraints.*

1.2. Our Approach

In line with our thesis statement, we investigate three major directions that enable us to make PIM effective and practical in computing systems: (1) examining the suitability of PIM across key workloads, (2) addressing major system challenges for adopting PIM in computing systems, and (3) redesigning applications to be aware of PIM capability. Toward these three major directions, we make four key contributions:

1.2.1. Identifying Key Primitives by Examining the Suitability of PIM Across Important Google Consumer Workloads

Our first work aims to identify important primitives for PIM by investigating the suitability of PIM across key mobile (consumer) workloads. We are experiencing an explosive growth in the number of consumer devices, including smartphones, tablets, web-based computers such as Chromebooks, and wearable devices. For this class of devices, energy efficiency is a first-class concern due to the limited battery capacity and thermal power budget. We find that *data movement* is a major contributor to the total system energy and execution time in consumer devices. The energy and performance costs of moving data between the memory system and the compute units are significantly higher than the costs of computation. As a result, addressing data movement is crucial for consumer devices.

In this work, we comprehensively analyze the energy and performance impact of data movement for several widely-used Google consumer workloads: (1) the Chrome web browser; (2) TensorFlow Mobile, Google's machine learning framework; (3) video playback, and (4) video capture, both of which are used in many video services such as YouTube and Google Hangouts. our comprehensive analysis of several popular Google consumer workloads shows that, among the many sources of energy consumption in consumer devices (e.g., CPUs, GPU, special-purpose accelerators, memory), data movement accounts for 62.7% of the total system energy. We find that *PIM* can significantly reduce data movement for all of these workloads, by performing part of the computation close to memory. Each workload contains simple primitives and functions that contribute to a significant amount of the overall data movement. We investigate whether these primitives and functions are feasible to implement using PIM, given the limited area and power constraints of consumer devices. Our analysis shows that offloading these primitives to PIM logic, consisting of either simple cores or specialized accelerators, eliminates a large amount of data movement, and significantly reduces total system energy (by an average of 55.4% across the workloads) and execution time (by an average of 54.2%). Chapter 4 describes this work in more detail.

1.2.2. Cache Coherence Support for Near-Data Accelerators (CoNDA)

Our second work aims to address the coherence challenge for *near-data accelerators* (NDAs). Recent advances in memory technology have enabled NDAs, which reside *off-chip* close to main memory and can yield further benefits than on-chip accelerators. Enabling coherence for NDAs provides two key benefits: (1) programmers can use the well-known traditional shared memory model to program systems with NDAs, and (2) we can simplify how NDAs communicate and share data with the rest of the system. However, enforcing coherence between NDAs and the rest of the system is very challenging. This is because (1) the cost of communication between NDAs and CPUs is high, and (2) NDA applications generate a lot of off-chip data movement. We find that existing coherence mechanisms eliminate most of the benefits of NDAs. We extensively analyze these mechanisms, and observe that (1) the majority of off-chip coherence traffic is unnecessary, and (2) much of the off-chip traffic can be eliminated if a coherence mechanism has insight into the memory accesses performed by the NDA.

Based on our observations, we propose 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 exploits this information to avoid performing *unnecessary* coherence requests, and thus, significantly reduces data movement for coherence. We evaluate CoNDA using state-of-the-art graph processing and hybrid in-memory database workloads. Averaged across all of our workloads operating on modest data set sizes, CoNDA improves performance by 19.6% over the highest-performance prior coherence mechanism (66.0%/51.7% over a CPU-only/PIM-only system) and reduces memory system energy consumption by 18.0% over the most energy-efficient prior coherence mechanism (43.7% over CPU-only). We find that CoNDA comes within 10.4% and 4.4% of the performance and energy of an ideal mechanism with no cost for coherence. The benefits of CoNDA increase with large data sets, as CoNDA improves performance over the highest-performance prior coherence mechanism by 38.3% (8.4x/7.7x over CPU-only/NDA-only), and comes within 10.2% of an *ideal* no-cost coherence mechanism. Chapter 5 describes CoNDA in more detail.

1.2.3. Mitigating Edge Machine Learning Inference Bottlenecks (Mensa)

Applications can fully benefit from PIM if both the hardware and software are co-designed to take advantage of PIM. Our third work aims to co-design hardware and software to make use of PIM for mobile machine learning applications to enable energy efficient and high performance inference execution. The demands of modern consumer devices are pushing machine learning (ML) inference to the network edge, with on-device computation. This has resulted in the design of edge ML accelerators that can compute a wide range of neural network (NN) models while still fitting within the tight resource constraints of edge devices. We analyze a commercial Edge TPU (tensor processing unit) using 24 Google edge NN models (including CNNs, LSTMs, transducers, and RCNNs), and find that the accelerator suffers from three shortcomings, in terms of computational throughput, energy efficiency, and memory access handling. We comprehensively study the characteristics of each NN layer in all of the Google edge models, and find that these shortcomings arise from the one-size-fits-all approach of the accelerator, as there is a high amount of heterogeneity in key layer characteristics both across different models and across different layers in the same model.

To combat this inefficiency, we propose a new acceleration framework called Mensa. Mensa incorporates multiple heterogeneous ML edge accelerators (including both on-chip and near-data accelerators), each of which caters to the characteristics of a particular subset of models. At runtime, Mensa schedules each layer to run on the best-suited accelerator, accounting for both efficiency and inter-layer dependencies. As we analyze the Google edge NN models, we discover that all of the layers naturally group into a small number of clusters, which allows us to design an efficient implementation of Mensa for these models with only three specialized accelerators. Averaged across all 24 Google edge models, Mensa improves energy efficiency and throughput by 3.0x and 3.1x over the Edge TPU, and by 2.4x and 4.3x over Eyeriss v2, a state-of-the-art accelerator. Chapter 6 describes Mensa design.

1.2.4. Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design (Polynesia)

Our fourth work aims to redesign emerging modern hybrid databases to be aware of PIM capability to enable real-time analysis. An exponential growth in data volume, combined with increasing demand for real-time analysis (i.e., using the most recent data), has resulted in the emergence of database systems that concurrently support transactions and data analytics. These *hybrid transactional and analytical processing* (HTAP) database systems can support real-time data analysis without the high costs of synchronizing across separate single-purpose databases. To support both types of workloads with high throughput and minimal energy, an ideal HTAP system should have three properties [255]. First, it should ensure that transactional and analytical workloads benefit from their own workload-specific optimizations (e.g., algorithms, data structures). Second, it should guarantee data freshness (i.e., access to the most recent version of data) for analytical workloads while ensuring that transactional and analytical workloads have a consistent view of data across the system. Third, it should ensure that the latency and throughput of transactional and analytical workloads are the same as if they were run in isolation.

Meeting all three ideal HTAP properties at once is very challenging, as transactional and analytical workloads have different underlying algorithms and access patterns, and optimizing for one property can often require a trade-off in another property. We extensively study state-of-the-art in-memory HTAP systems and find that no system exists that can meet all three properties. We observe two key problems that prevent state-of-the-art HTAP systems from simultaneously achieving the three desired properties. First, these systems experience a drastic reduction in transactional throughput (up to 74.6%) and analytical throughput (up to 49.8%) compared to when we run each in isolation. This is because the mechanisms used to provide data freshness and consistency induce a significant amount of *data movement* between the CPU cores and main memory. Second, HTAP systems often fail to provide effective performance isolation. These systems suffer from severe performance interference (up to 31.3% reduction in transactional throughput) because of the high resource contention between transactional workloads and analytical workloads.

To solve the challenges faced by existing HTAP systems, we propose a novel hardware/software cooperative design for in-memory HTAP databases called Polynesia. The key idea of Mensa is to partition the computing resources in a system into two types of isolated, specialized processing *islands* (*transactional islands* and *analytical islands*). By isolating transactional islands from analytical islands, we are able to (1) apply workload-specific optimizations to each island; (2) avoid high resource contention; and (3) design new and efficient mechanisms that propagate data updates from transactional islands to analytical islands, where we can provide data freshness and consistency without incurring high data movement costs. Polynesia (1) divides the HTAP system into transactional and analytical processing islands, (2) implements custom algorithms and hardware to reduce the costs of update propagation and consistency, and (3) exploits processing-in-memory for the analytical islands to alleviate data movement. Our evaluation shows that Polynesia outperforms three state-of-the-art HTAP systems, with average transactional/analytical throughput improvements of 1.7X/3.7X, and reduces energy consumption by 48% over the prior lowest-energy system. Chapter 7 describes the Polynesia design.

1.3. Contributions

This dissertation makes the following major contributions:

- This dissertation provides the first comprehensive analysis of important Google consumer workloads, including the Chrome browser [112], TensorFlow Mobile [128], video playback [134], and video capture [134], to identify major sources of energy consumption. We observe that data movement between the main memory and conventional computation units is a major contributor to the total system energy consumption in consumer devices. On average, data movement accounts for 62.7% of the total energy consumed by Google consumer workloads. We observe that most of the data movement in consumer workloads is generated by simple functions and primitives.

- This dissertation presents the first detailed analysis of the feasibility of PIM for consumer devices, considering the stringent power and area constraints of such devices. Our evaluation shows that we can design cost-efficient PIM logic that significantly reduces the total system energy and execution time of consumer workloads, by 55.4% and 54.2%, respectively, averaged across all of the consumer workloads we study. These functions and primitives are composed of operations such as *memcopy*, *memset*, basic arithmetic operations, and bitwise operations, all of which can be implemented in hardware at low cost.

- This dissertation provides an extensive design exploration and show that (1) the poor handling of coherence eliminates much of an NDA's performance and energy benefits, (2) the majority of off-chip data movement (i.e., coherence traffic) generated by a coherence mechanism is unnecessary, and (3) a significant portion of unnecessary coherence traffic can be eliminated by having insight into which memory accesses actually require a coherence operation.

- This dissertation proposes CoNDA, a new coherence mechanism that *optimistically executes* code on an NDA to gather information on memory accesses. Optimistic execution enables CoNDA to identify and avoid performing *unnecessary* coherence requests. As our evaluation shows, this reduces off-chip data movement, allowing NDA execution under CoNDA to always outperform prior coherence mechanisms (as well as CPU-only and NDA-only execution).

- This dissertation presents the first in-depth analysis of how edge ML accelerators operate across a wide range of state-of-the-art edge NN models (provided by Google). Our analysis reveals three key shortcomings of state-of-the-art accelerators: (1) significant PE underutilization, (2) poor energy efficiency, and (3) an inefficient memory system. We comprehensively analyze the key characteristics of each layer in Google's edge NN models. We make two observations from our analysis: (1) layer characteristics vary significantly both *across* models and across layers *within a single model*, and (2) the monolithic design of state-of-the-art accelerators is the root cause of their shortcomings.

- This dissertation proposes Mensa, a new framework for efficient edge ML acceleration. Mensa employs a few small, carefully specialized accelerators, and includes a runtime scheduler that assigns each layer to one of the accelerators based on layer characteristics and inter-layer communication. We find that Mensa is significantly more efficient and performs better than a state-of-the-art edge ML accelerator for our Google edge models, while using significantly less area.
- This dissertation provides the first comprehensive analysis of how major system- and architecture-level challenges limit throughput and efficiency in state-of-the-art HTAP systems.
- This dissertation proposes Polynesia, a new hardware/software cooperative design for HTAP databases. We isolate hardware resources into transactional and analytical islands, and design new algorithms and specialized hardware for running analytical workloads and for conveying transactional updates to analytical workloads. Polynesia is the first work to achieve all three desired properties of an HTAP system. We propose new algorithms for update propagation, consistency, data placement, and task scheduling, which we co-design with new hardware to take advantage of the underlying characteristics of HTAP workloads. To our knowledge, this is the first specialized hardware for HTAP databases.

1.4. Dissertation Outline

This dissertation is organized into 8 chapters. Chapter 2 presents background on 3D-stacked memory and discusses early PIM proposals. Chapter 3 discusses related prior work on PIM. Chapter 4 presents our comprehensive analysis of Google consumer workloads and detailed analysis of the feasibility of PIM for consumer devices. Chapter 5 presents the design of CoNDA, our cache coherence mechanism for PIM. Chapter 6 presents our comprehensive analysis of edge ML accelerators and Google NN models, and discusses the design of Mensa, our proposed framework for efficient edge ML acceleration. Chapter 7 discusses the major system- and architecture-level challenges of state-of-the-art HTAP systems and presents Polynesia, our hardware/software cooperative design for HTAP databases. Finally, Chapter 8 presents conclusions and future research directions that are enabled by this dissertation.

Chapter 2

Background

2.1. Early PIM Proposals

The origins of PIM go back to proposals from the 1970s, where small processing elements were combined with small amounts of RAM to provide a distributed array of memories that perform computation [333, 347]. Starting in the 1990s, many different groups of researchers explored the integration of logic close to the memory subsystem. Some of the early works on PIM [81,109,218,293] add logic within DRAM to perform vector operations. Execube is among very early PIM implementation which targets massively parallel workloads. It integrates 8 processing cores (16-bit SIMD/MIMD) with a memory bank (4 Mbits of DRAM), and connect all processing elementrs together to provide direct memory access transfers between them. IRAM [293] is another early PIM implementation which propose to combine 13 MB of DRAM with a vector processor on a single chip, primarily targeting multimedia workload. Computationl-RAM [81] and Terasys [109] propose to integrate a large array of simple computation directly into DRAM arrays to exploit the massive internal bandwidth of DRAM.

Later works [76, 199, 254, 267, 287] propose more versatile substrates that increase the flexibility and computational capability available within the DRAM chip. FlexRAM [199] serves as a replacement for the main DRAM system, with logic and memory chips integrated together to work as a coprocessor. DIVA [76] and Smart Memories [254] are alternative architectures that combine memory chips with sophisticated processor components (e.g., functional units, full caches) to act as independent processors. The Active Pages proposal [287] uses a reconfigurable architecture to make memory more flexible, and issues computations to memory that can be performed across all of the data within a large page. Unfortunately, many of these works were hindered by the limitations of existing memory technologies and costly DRAM–logic integration, which prevented the practical integration of logic in or near the memory.

2.2. New Opportunities in Modern Memory Systems

Due to the increasing need for large memory systems by modern applications, DRAM scaling is being pushed to its practical limits [197]. It is becoming more difficult to increase the density citedram-scaling, drame-latency, reduce the latency [211, 273], and decrease the energy consumption of conventional DRAM architectures. In response, memory manufacturers are actively developing two new approaches for main memory system design, both of which can be exploited to overcome prior barriers to implementing PIM architectures.

The first major innovation is 3D-stacked memory [233, 250]. The emergence of 3D-stacked DRAM architectures [174, 186, 233, 250] offers a promising solution to enable PIM. These architectures stack multiple layers of DRAM arrays within a single chip, as shown in Figure 2.1. These 3D-stacked memory use *through-silicon vias* (TSVs), which are vertical wires that connect all stack layers together. With current manufacturing process technologies, thousands of TSVs can be placed within a single 3D-stacked memory chip. Due to the available density of TSVs, 3D-stacked memories are able to provide much greater bandwidth between stack layers than they can provide off-chip. Examples of 3D-stacked DRAM available commercially include High-Bandwidth Memory (HBM) [186], Wide I/O [189], and the Hybrid Memory Cube (HMC) [174].

Several 3D-stacked DRAM architectures (e.g., HBM [186], HMC [174]) provide a dedicated *logic layer* within the stack that can have low-complexity (due to thermal constraints) logic. The logic layer is typically the bottommost layer of the chip, and is connected to the same TSVs as the memory layers. The logic layer provides a space inside the DRAM chip where architects can implement functionality that interacts with both the processor and the DRAM cells. Currently, manufacturers make limited use of the logic layer, presenting an opportunity for architects to

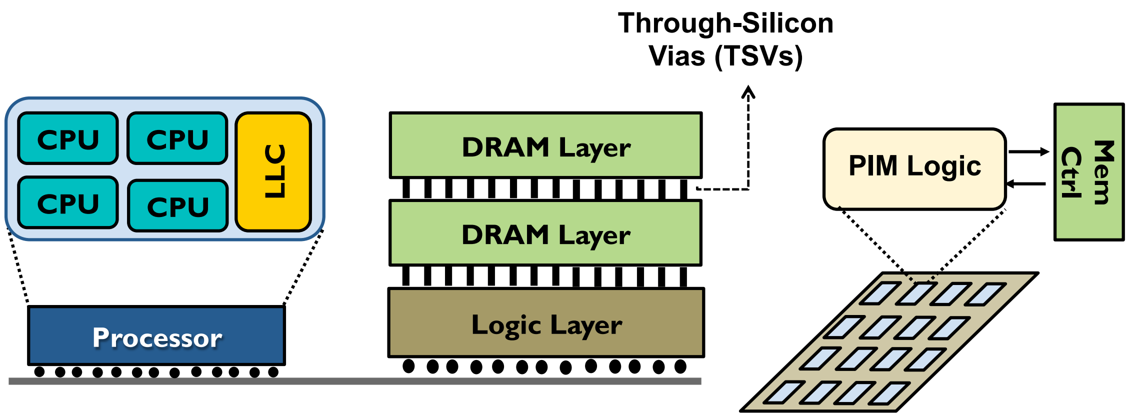


Figure 2.1. High-level organization of a 3D-stacked DRAM archiecture.

implement new PIM logic in the available area of the logic layer. We can potentially add a wide range of computational logic (e.g., general-purpose cores, accelerators, reconfigurable architectures) in the logic layer, as long as the added logic meets area, energy, and thermal dissipation constraints.

Recent PIM proposals [13, 14, 15, 42, 77, 93, 97, 99, 139, 167, 169, 207, 248, 275, 319, 320, 321, 342, 356, 381, 389, 392] add processing logic to the logic layer to exploit the high bandwidth available between the logic layer and the DRAM cell arrays. The proposed PIM processing logic design varies based on the specific architecture, and can range from fixed-function accelerators to simple in-order cores, and reconfigurable logic. The complexity of the processing logic that can be added to the logic layer is currently limited by the manufacturing process technology and thermal design points, which may prevent highly-sophisticated processors (e.g., out-of-order processor cores with large caches and sophisticated instruction-level parallelism techniques) from being implemented within the logic layer at this time.

The second major innovation is the use of byte addressable resistive nonvolatile memory (NVM) for the main memory subsystem. In order to avoid DRAM scaling limitations entirely, researchers and manufacturers are developing new memory devices that can store data at much higher densities than the typical density available in existing DRAM manufacturing process technologies. Manufacturers are exploring at least three types of emerging NVMs to augment or replace DRAM at the main memory layer: (1) phase-change memory (PCM), (2) magnetic RAM (MRAM) and (3) metal-oxide resistive RAM (RRAM) or memristors. All three of these NVM types are expected to provide memory access latencies and energy usage that are competitive with or close enough to

DRAM, while enabling much larger capacities per chip and nonvolatility in main memory

NVMs present architects with an opportunity to redesign how the memory subsystem operates. While it can be difficult to modify the design of DRAM arrays due to the delicacy of DRAM manufacturing process technologies as we approach scaling limitations, NVMs have yet to approach such scaling limitations. As a result, architects can potentially design NVM memory arrays that integrate PIM functionality. A promising direction for this functionality is the ability to manipulate NVM cells at the circuit level in order to perform logic operations using the memory cells themselves. A number of recent works have demonstrated that NVM cells can be used to perform a complete family of Boolean logic operations [24, 242]. similar to such operations that can be performed in DRAM cells [319, 321, 369].

2.3. Processing-Near-Memory vs. Processing-Using-Memory

Many recent works take advantage of the memory technology innovations that we discuss in Section 2.2 to enable PIM. We find that these works generally take one of two approaches: (1) processing-near-memory, and (2) processing-using-memory. Processing-near-memory involves adding or integrating PIM logic (e.g., accelerators, very small in-order cores, reconfigurable logic) close to or inside the memory ([14, 15, 16, 44, 77, 97, 99, 139, 144, 167, 168, 206, 210, 260, 261, 275, 294, 356, 380, 381, 389]). Many of these works place PIM logic inside the logic layer of 3D-stacked memories or at the memory controller. In contrast, processing-using-memory makes use of intrinsic properties and operational principles of the memory cells and cell arrays themselves, by inducing interactions between cells such that the cells and/or cell arrays can perform computation. Prior works show that processing-using-memory is possible using static RAM (SRAM) [12, 80], DRAM [71, 96, 241, 319, 320, 321, 322, 323, 324], or NVM technology [24, 35, 59, 145, 183, 242, 326, 355].

Chapter 3

Related Work

Many previous works have proposed to use PIM to mitigate data movement bottlenecks. We will describe the relevant works by dividing them into different categories based on their similarity.

3.1. Early Approaches to PIM

The origins of PIM go back to proposals from the 1970s, where small processing elements were combined with small amounts of RAM to provide a distributed array of memories that perform computation [333, 347]. Starting in the 1990s, many different groups of researchers explored the integration of logic close to the memory subsystem [81, 109, 218, 293]. Later works [76, 199, 254, 267, 287] propose more versatile substrates that increase the flexibility and computational capability available within the DRAM chip. Unfortunately, many of these works were hindered by the limitations of existing memory technologies and costly DRAM–logic integration, which prevented the practical integration of logic in or near the memory. We provide more details on these early works in Chapter 2.

3.2. PIM Using 3D-Stacked Memories

With the advent of 3D-stacked memories, we have seen a resurgence of PIM proposals [14, 15, 16, 31, 44, 48, 77, 94, 97, 98, 99, 137, 139, 144, 152, 154, 167, 168, 206, 210, 236, 260, 261, 275, 294, 310, 341, 356, 380, 381, 389, 395, 400]. Recent PIM proposals add compute units within the logic layer to exploit the high bandwidth available. These works primarily focus on the design of the underlying logic that is placed within memory, and in many cases propose special-purpose PIM architectures that cater only to a limited set of applications. These works include accelerators for data reorganization [16], graph processing [14, 275, 392], databases [77, 261, 380], in-memory analytics [97], MapReduce [302], genome sequencing [210], graphics [381], machine learning workloads [99, 206], and concurrent data structures [249]. Some works propose more generic architectures by adding PIM-enabled instructions [15], GPGPUs [167, 294, 389], or reconfigurable hardware [93, 139] to the logic layer in 3D-stacked memory. More work in the area can be found in the following four major overview papers [103, 271, 272, 325].

3.3. PIM Architectures for Consumer Workloads

Recent works on PIM embed computation units within the logic layer (e.g., [14, 15, 16, 44, 77, 97, 99, 139, 144, 167, 168, 206, 210, 260, 261, 275, 294, 356, 380, 381, 389]). However, these previous proposals are *not* designed for the highly-stringent area, power, and thermal constraints of modern commercial consumer devices. To our knowledge, this dissertation presents the first work (Chapter 4) to (1) conduct a comprehensive analysis of Google's consumer device workloads to identify major sources of energy consumption, with a focus on data movement; and (2) analyze and evaluate how PIM benefits consumer devices, given the stringent power and area constraints of such devices.

Prior works [50, 142, 171, 289, 309] study the general performance and energy profile of mobile applications. Narancic et al. [276] study the memory system behavior of a mobile device. Shingari et al. [335] characterize memory interference for mobile workloads. None of these works analyze (1) the sources of energy consumption and data movement in consumer workloads, or (2) the benefits of PIM for modern consumer workloads. One prior work [290] measures the data movement cost of emerging mobile workloads. However, the work does not identify or analyze the sources of data movement in consumer workloads, and does not propose any mechanism to reduce data movement.

A number of prior works select a single consumer application and propose techniques to improve the application. Many prior works [53, 54, 397, 399] use a variety of techniques, such as microarchitecture support and language extensions, to improve browsing energy and performance. None of these works (1) conduct a detailed analysis of important user interactions (page scrolling and tab switching) when browsing real-world web pages, or (2) identify sources of energy consumption and data movement during these interactions. Other works [45, 61, 70, 192, 229, 232, 390] attempt to improve the efficiency of motion compensation, motion estimation, and the deblocking filter in different video codec formats using various techniques, ranging from software optimization to hardware acceleration. None of these works (1) thoroughly analyze the data movement cost for a video codec, or (2) use PIM to improve the energy and performance of the video codec. Several works [58, 99, 151, 206, 307] focus on accelerating inference. While these mechanisms can speed up inference on mobile TensorFlow, they do *not* address the data movement issues related to packing and quantization. Tetris [99] and Neurocube [206] attempt to push the *entire* inference execution into PIM. While Tetris and Neurocube eliminate data movement during inference, their PIM logic incurs high area and energy overheads, which may not be practical to implement in consumer devices given the *limited* area and power budgets. In contrast, our approach identifies the *minimal* functions of inference that benefit from PIM, allowing us to greatly reduce data movement without requiring a large area or energy overhead, and making our PIM logic feasible to implement in consumer devices.

3.4. Cache Coherence Support for PIM

Most recent PIM proposals assume that there is only a limited amount of data sharing between PIM kernels and the CPU threads, and employ naive solutions for coherence, such as (1) assuming that data is never accessed concurrently [14, 169], (2) making PIM data non-cacheable in the CPU [14, 77, 93, 99, 275, 302], (3) flushing *all* dirty cache lines from CPU caches before starting an PIM kernel [97, 235, 361, 389], (4) using coarse-grained coherence [93, 380], or (5) using traditional fine-grained coherence protocols [42]. We showe in this dissertation (Chapter 5) that these approaches are not effective for several important application domains. To our knowledge, this dissertation presents the first work (Chapter 5) to (1) perform an extensive design exploration of state-of-the-art mechanisms for PIM–CPU coherence, and (2) propose an efficient mechanism for coherence at a fine granularity between PIM kernels and CPU threads.

Other works aim to provide efficient coherence support for accelerators in non-PIM systems. HSC [298] reduces coherence traffic between the CPU and GPU. However, HSC assumes that both the CPU and GPU are on the same chip, and thus can benefit from CG, unlike many PIM systems. FUSION [222] employs MESI (fine-grained, or FG) coherence between on-chip accelerators and the CPU. As we show in this dissertation, FG eliminates the majority of the benefits of an PIM due to the high cost of *off-chip* traffic.

Several works [20, 204, 286, 340] reduce on-chip communication between GPU cores, and are also not well-suited for PIM–CPU coherence. For example, the DeNovo protocol [340] communicates at a fine granularity when (1) a cache miss occurs, (2) the protocol registers a write (i.e., informs the directory of the core that holds the modified data), or (3) a cache line is evicted. Since PIM kernels often have poor cache locality, this would cause DeNovo to generate a large amount of off-chip traffic. Furthermore, protocols like DeNovo are not readily compatible with existing coherence protocols, and require us to implement the new protocol across the entire system [340], which is a barrier to adoption. In contrast, CoNDA is designed with the poor cache locality of PIM kernels in mind, and does not require the modification of existing coherence protocols elsewhere in the system. We believe these works [20, 204, 286, 340] can be used to optimize *intra*-PIM coherence.

3.5. Software–Hardware Co-design for PIM

Many recent works propose to use PIM to mitigate data movement costs. In particular, serveral prior works propose PIM architectures for certain application domain such as data analytic system (e.g., [77,261,302,380]), deep learning application (e.g, [97,99,206]) and graph processing systems (e.g., [14,97,275,392]). Only a few of these works propose software-hardware co-design techniques aware of PIM capability. Mondrian [77] proposes to modify the partitioning algorithm in the partitioning phase of data analytics operators to improve the efficiency of memory access patterns for PIM. GraphP [392] proposes a software-hardware co-designed graph processing system which features a new partitioning algorithm, programming model, and synchronization algorithm that make use of HMC and PIM architecture. However, no prior work proposes software-hardware co-design techniques for HTAP or mobile ML applications.

3.6. HTAP Systems

Several works from industry (e.g., [92, 107, 227, 230, 231]) and academia (e.g., [25, 30, 136, 205, 215, 234, 255, 266, 314]) propose various techniques to support HTAP. Many of them use a single-instance design [25, 30, 92, 136, 205, 314], For example, Hyper [205] proposes a single replica with NSM data layout, and uses virtual memory system to provide analytics with snapshots of the most recent version of data, while transactions update the main replica. SAP HANA [92] stores data in a main–delta structure, where updates are stored in a delta structure and are periodically merged back into the main partition. Peloton [30] proposes a hybrid data layout and periodically converts data between different formats to benefit both analytical and transactional workloads. H2TAP [25] proposes to move the analytical workload from the CPU to a GPU, and uses shared memory and message passing to provide data freshness and consistency. Many other works use a multiple-instance design [5, 107, 255]. Batch-DB [255] uses two replicas, one for transactions and the other for analytics, and propagates updates between them. Both Oracle [5] and the scale-out extension of SAP-HANA [107] use a delta structure and periodically merge updates from the transactional replica into the analytical replica. All of these proposals suffer from the drawbacks we highlight in Chapter 1.2.4 (and provides more detail in Chapter 7), and none can fully meet the desired HTAP properties. Other prior works focus solely on analytical workloads [77, 217, 261, 377, 378, 380]. Some of these works propose to use specialized on-chip accelerators [217, 377, 378] while others propose to use PIM to speed up analytical operators [77, 261, 380]. However, none of these works study the effect of data placement or task scheduling for the analytical workload in the context of PIM or HTAP systems.

To our knowledge, this dissertation presents the first work (Chapter 7) that (1) comprehensively examines HTAP systems and their major challenges, (2) proposes a hardware–software co-designed HTAP system, and (3) describes an HTAP system that meets all desired HTAP properties.

3.7. Edge ML Inference

Many prior works look at a specific model type (predominantly CNNs) [18, 21, 56, 57, 58, 78, 99, 138, 141, 201, 251, 292, 328, 334, 338, 362, 365, 394]. None of these works perform an analysis across different classes of edge models (e.g., Transducers, LSTMs, RCNNs). In fact, many of these works (e.g., [58, 78, 99, 292, 328]) analyze traditional models (e.g., AlexNet, VGG), and their proposals are not tailored toward state-of-the-art edge models (which we show are different) or resource-limited edge devices. The proposed accelerators are tailored toward a particular model type (e.g., CNNs [58, 99, 292, 334, 362], LSTMs [338, 394]), and they are not optimized to serve multiple edge model types. As a result, these accelerators all suffer from the issues we discuss in Chapter 6.

Among those works that focus on CNN models, a few observe diversity across CNN layers [57, 198, 224, 225, 328, 365]. Maeri [225] and EyerissV2 [57] use reconfigurable logic to provide flexible dataflows and networks for different layers. However, those solutions (1) are very costly for area- and power-constraint edge devices, and (2) do not address memory system issues for edge accelerators. ScaleDeep [365] proposes customized processing tiles to address diversity in CNN layers. While the idea shares some similarities with our proposed approach (Mensa in Chaptern 6) in this dissertation (exploiting heterogeneity in hardware), ScaleDeep (1) targets traditional cloud-based training instead of edge inference, resulting in significantly different support for diversity (e.g., CNN layers in edge models have significantly greater diversity than traditional models); and (2) does not address the larger diversity that exists between CNN layers and layers in other model types (e.g., LSTM layers). Neurosurgeon [198] looks at both vision and speech models. However, their analysis is done on old/traditional vision/speech models (which do not reflect state-of-the-art edge model properties), and their solution relies on offloading some layers to the cloud, which undermines the goal of running inference locally.

To the best of our knowledge, this dissertation presents the first work (mensa in Chapter 6) that (1) extensively analyzes a wide range of state-of-the-art Google edge models and how a state-of-the-art edge accelerator operates across these models, (2) quantify the large degree of layer variation both across edge models and within a model, and (3) proposes a framework to efficiently manage multiple heterogeneous edge ML accelerators that exploit layer variation.

3.8. Addressing Challenges to PIM Adoption

Recent work has examined design challenges for systems with PIM support that can affect PIM adoption. These challenges are being worked on by the research community and industry to facilitate PIM adoption into conventional computing systems [103, 271, 272, 325]. A number of these works improve PIM programmability, such as the study by Sura et al. [352], which optimizes how programs access PIM data; PEI [15], which introduces an instruction-level interface for PIM that preserves the existing sequential programming models and abstractions for virtual memory and coherence; TOM [167], which automates the identification of basic blocks that should be offloaded to PIM and the data mapping for such blocks; work by Pattnaik et al. [294], which automates whether portions of GPU applications should be scheduled to run on GPU cores or PIM cores; and work by Liu et al. [249], which designs PIM-specific concurrent data structures to improve PIM performance. Other works tackle hardware-level design challenges, including IMPICA [168], which introduces in-memory support for address translation and pointer chasing; and work by Kim et al. [208] that enables PIM logic to efficiently access data across multiple memory stacks. There is a recent work on modeling and understanding the interaction between programs and PIM hardware, such as NAPEL [342], a framework that predicts the potential performance and energy benefits of using PIM.

Chapter 4

Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks

Consumer devices, which include smartphones, tablets, web-based computers such as Chromebook [113], and wearable devices, have become increasingly ubiquitous in recent years. There were 2.3 billion smartphone users worldwide in 2017 [83]. Tablets have witnessed similar growth, as 51% of the U.S. population owns a tablet as of 2016, and 1.2 billion people in the world use tablets [83]. There is similar demand for web-based computers like Chromebooks [113], which now account for 58% of all computer shipments to schools in the U.S. [159].

Energy consumption is a first-class concern for consumer devices. The performance requirements of these consumer devices have increased dramatically every year to support emerging applications such as 4K video streaming and recording [391], virtual reality (VR) [228], and augmented reality (AR) [17, 55, 146, 247]. A consumer device integrates many power-hungry components such as powerful CPUs, a GPU, special-purpose accelerators, sensors, and a high-resolution screen. Despite the rapid growth in processing capability, two trends greatly limit the performance of consumer devices. First, lithium-ion battery capacity has only doubled in the last 20 years [69, 312]. Second, the thermal power dissipation of consumer devices has become a severe performance constraint [146]. Therefore, fundamentally energy-efficient design of consumer devices is critical to keep up with increasing user demands and to support a wide range of emerging applications [55, 210, 228, 262, 264, 265, 280, 391].

Our goal is to (1) understand the data movement related bottlenecks in modern consumer workloads, (2) comprehensively analyze the benefits that PIM can provide for such workloads, and (3) investigate the PIM logic that can benefit these workloads while still being feasible to implement given the limited area, power, and thermal budgets of consumer devices.

4.1. Summary of Key Insights and Observations

To identify the major sources of energy consumption in consumer devices, we conduct an in-depth analysis of several popular Google consumer workloads, as they account for a significant portion of the applications executed on consumer devices. We analyze (1) Chrome [112], the most commonly-used web browser [277]; (2) TensorFlow Mobile [128], Google's machine learning framework that is used in various services such as Google Translate [124], Google Now [123], and Google Photos [121]; (3) video playback [134] and (4) video capture using the VP9 codec [134], which is used by many video services such as YouTube [130], Skype [259], and Google Hangouts [120]. These workloads are among the most commonly-used applications by consumer device users [68, 73, 85, 171, 289], and they form the core of many Google services (e.g., Gmail [117], YouTube [130], the Android OS [111], Google Search [122]) that each have over a billion monthly active users [131, 297].

We make a **key observation** based on our comprehensive workload analysis: among the many sources of energy consumption in consumer devices (e.g., CPUs, GPU, special purpose accelerators, memory), *data movement* between the main memory system and computation units (e.g., CPUs, GPU, special-purpose accelerators) is a major contributor to the total system energy. For example, when the user scrolls through a Google Docs [119] web page, moving data between memory and computation units causes 77% of the total system energy consumption (Section 4.3.2.1). This is due to the fact that the energy cost of moving data is orders of magnitude higher than the energy cost of computation [202]. We find that across all of the applications we study, 62.7% of the total system energy, on average, is spent on data movement between main memory and the compute units.

Based on our key observation, we find that we can substantially reduce the total system energy if we greatly mitigate the cost of data movement. One potential way to do so is to execute the data-movement-heavy portions of our applications *close to the data*. Recent advances in 3D-stacked memory technology have enabled cost-effective solutions to realize this idea [174,186,233,250]. 3D-stacked DRAM architectures include a dedicated *logic layer*, that is capable of providing logic functionality, with high-bandwidth low-latency connectivity to DRAM layers. Recent works [14,15, 16,44,77,97,99,139,144,167,168,206,210,260,261,275,294,356,380,381,389] take advantage of the logic layer [174,186,233,250] to perform *processing-in-memory* (PIM), also known as *near-data processing*. PIM allows the CPU to dispatch parts of the application for execution on compute units that are close to DRAM. Offloading computation using PIM has two major benefits. First, it eliminates a significant portion of the data movement between main memory and conventional processors. Second, it can take advantage of the high-bandwidth and low-latency access to the data inside 3D-stacked DRAM.

However, there are challenges against introducing PIM in consumer devices. Consumer devices are extremely stringent in terms of the area and energy budget they can accommodate for any new hardware enhancement. Regardless of the memory technology used to enable PIM (whether it is HMC [174], HBM [186], or some other form of 3D-stacked or compute-capable memory [214, 233,319,321]), any additional logic can potentially translate into a significant cost in consumer devices. In fact, unlike prior proposals for PIM in server or desktop environments, consumer devices may not be able to afford the addition of full-blown general-purpose PIM cores [44,77,97], GPU PIM cores [167,294,389], or sophisticated PIM accelerators [14,99,168] to 3D-stacked memory. As a result, a major challenge for enabling PIM in consumer devices is to identify what kind of in-memory logic can both (1) *maximize energy efficiency* and (2) be implemented at *minimum possible cost*.

To investigate the potential benefits of PIM, given the area and energy constraints of consumer devices, we delve further into each consumer workload to understand what underlying functions and characteristics contribute most to data movement. Our analysis leads to a **second key observation**: across all of the consumer workloads we examine, there are often simple functions and primitives (which we refer to as *PIM targets*) that are responsible for a significant fraction of the total data movement. These PIM targets range from simple data reorganization operations, such as tiling and packing, to value interpolation and quantization. For example, as we show in Sections 4.3 and 4.4, the data movement cost of data reorganization operations in Chrome and TensorFlow Mobile account for up to 25.7% and 35.3% of the total system energy, respectively.

We find that many of these PIM targets are comprised of simple operations such as *memcopy, memset*, and basic arithmetic and bitwise operations. Such PIM targets are mostly data-intensive and require relatively little and simple computation. For example, texture tiling in Chrome is comprised of *memcopy*, basic arithmetic, and bitwise operations. We find that the PIM targets can be implemented as PIM logic using either (1) a small low-power general-purpose embedded core (which we refer to as a *PIM core*) or (2) a group of small fixed-function accelerators (*PIM accelerators*). Our analysis shows that the area of a PIM core and a PIM accelerator take up no more than 9.4% and 35.4%, respectively, of the area available for PIM logic in an HMC-like [174] 3D-stacked memory architecture (Section 4.2.2). Thus, the PIM core and PIM accelerator are cost-effective to use in consumer devices.

Our comprehensive experimental evaluation shows that PIM cores are sufficient to eliminate a majority of data movement, due to the computational simplicity and high memory intensity of the PIM targets. On average across all of the consumer workloads that we examine, PIM cores provide a 49.1% energy reduction (up to 59.4%) and a 44.6% performance improvement (up to 2.2x) for a state-of-the-art consumer device. We find that PIM accelerators provide larger benefits, with an average energy reduction of 55.4% (up to 73.5%) and performance improvement of 54.2% (up to 2.5x) for a state-of-the-art consumer device. However, PIM accelerators require custom logic to be implemented for each separate workload. We find that PIM accelerators are especially effective for workloads such as video playback, which already make use of specialized hardware accelerators in consumer devices.

4.2. Analyzing and Mitigating Data Movement

We start by identifying those portions of the consumer workloads that cause significant data movement and that are best suited for PIM execution. We examine four widely-used Google consumer workloads: (1) the Chrome web browser [112]; (2) TensorFlow Mobile [128], Google's machine learning framework; (3) video playback, and (4) video capture using the VP9 video codec [134]. These workloads form the core of many Google services (e.g., Gmail [117], YouTube [130], the Android OS [111], Google Search [122]) that each have over a billion monthly active users [131, 297].

In this section, we discuss our characterization methodology (Section 4.2.1), our approach for identifying PIM targets in each consumer workload (Section 4.2.2), and the types of PIM logic we use to implement the PIM targets (Section 4.2.3).

4.2.1. Workload Analysis Methodology

We perform our workload characterization on a Chromebook [113] with an Intel Celeron (N3060 dual core) SoC [177] and 2GB of DRAM. Our performance and traffic analyses are based on hardware performance counters within the SoC.

We build our energy model based on prior work [290], which sums up the total energy consumed by the CPU cores, DRAM, off-chip interconnects, and all caches. We use hardware performance counters to drive this model. During our characterization, we turn off Wi-Fi and use the lowest brightness for the display to ensure that the majority of the total energy is spent on the SoC and the memory system [289, 290]. We use CACTI-P 6.5 [270] with a 22nm process technology to estimate L1 and L2 cache energy. We estimate the CPU energy based on prior works [290, 364] and scale the energy to accurately fit our Celeron processor. We model the 3D-stacked DRAM energy as the energy consumed per bit, using estimates and models from prior works [184, 313]. We conservatively use the energy of the ARM Cortex-R8 to estimate the energy consumed by a PIM core, and we estimate the PIM accelerator energy based on [9], conservatively assuming that the accelerator is 20x more energy-efficient than the CPU cores.

Chrome Web Browser. We analyze Chrome [112] using the Telemetry framework [65], which automates user actions on a set of web pages. We analyze three pages that belong to important Google web services (Google Docs [119], Gmail [117], Google Calendar [118]), two of the top 25 most-accessed web sites [19] (WordPress [375] and Twitter [363]), and one animation-heavy page [65].

TensorFlow Mobile. To analyze TensorFlow Mobile [128], we profile four representative neural networks: VGG-19 [339], ResNet-v2-152 [157], Inception-ResNet-v2 [354], and Residual-GRU [360]. The first three networks are for image classification, and Residual-GRU is for image compression. All four networks have a number of use cases on mobile consumer devices (e.g., grouping similar images, reducing the size of newly-taken photos on the fly).

Video Playback and Video Capture. We evaluate both hardware and software implementations of the VP9 codec [134, 372]. We use publicly-available video frames [382] as inputs to the VP9 decoder and encoder. For the in-house hardware implementation, we use a bit-level C++ model to accurately model all traffic between each component of the hardware and DRAM. The RTL for the commercial VP9 hardware is generated from this C++ model using Calypto Catapult [258].

4.2.2. Identifying PIM Targets

We use hardware performance counters and our energy model to identify candidate functions that could be PIM targets. A function is a PIM target candidate if (1) it consumes the most energy out of the all functions in the workload, (2) its data movement consumes a significant fraction of the total workload energy, (3) it is memory-intensive (i.e., its last-level cache *misses per kilo instruction*, or MPKI, is greater than 10 [63, 212, 213, 268]), and (4) data movement is the single largest component of the function's energy consumption. We then check if each candidate is amenable to PIM logic implementation using two criteria. First, we discard any PIM targets that incur any performance loss when run on simple PIM logic (i.e., PIM core, PIM accelerator). Second, we discard any PIM targets that require more area than is available in the logic layer of 3D-stacked memory (see Section 4.2.3). In the rest of this chapter, we study only PIM target candidates that pass both of these criteria.

4.2.3. Implementing PIM Targets in 3D-Stacked DRAM

For each workload, once we have identified a PIM target, we propose PIM logic that can perform the PIM target functionality inside the logic layer of 3D-stacked memory. We propose two types of PIM logic: (1) a general-purpose *PIM core*, where a single PIM core can be used by *any* of our PIM targets; and (2) a fixed-function *PIM accelerator*, where we design custom logic for each PIM target.

For the PIM core, we design a custom 64-bit low-power single-issue core similar in design to the ARM Cortex-R8 [28] (see Section 4.8 for more details). All of the PIM targets that we evaluate are data-intensive, and the majority of them perform only simple operations (e.g., *memcopy*, basic arithmetic operations, bitwise operations). As a result, we do *not* implement aggressive instruction-level parallelism (ILP) techniques (e.g., sophisticated branch predictors, superscalar execution) in our core. Several of our PIM targets exhibit highly *data-parallel* behavior, leading us to incorporate a SIMD unit that can perform a single operation on multiple pieces of data concurrently. We empirically set the width of our SIMD unit to 4.

For the PIM accelerator for each of our PIM targets, we first design a customized *in-memory logic unit*, which is fixed-function logic that performs a single thread of the PIM target in the logic layer. To exploit the data-parallel nature of the PIM targets, we add multiple copies of the in-memory logic unit, so that multiple threads of execution can be performed concurrently.

We evaluate the performance and energy consumption of a state-of-the-art consumer device that contains either PIM cores or PIM accelerators in 3D-stacked memory (Section 4.9). In order to assess the feasibility of implementing our PIM cores or PIM accelerators in memory, we estimate the area consumed by both types of PIM logic for a 22 nm process technology. We assume that the 3D-stacked memory contains multiple *vaults* (i.e., vertical slices of 3D-stacked DRAM), and that we add one PIM core or PIM accelerator per vault. Assuming an HMC-like [174] 3D-stacked memory architecture for PIM, there is around 50–60 mm^2 of area available for architects to add new logic into the DRAM logic layer. This translates to an available area of approximately 3.5–4.4 mm^2 *per vault* to implement our PIM logic [77, 99, 184].[1] We find that each PIM core requires less than 0.33 mm^2 of area in the logic layer, conservatively based on the footprint of the ARM Cortex-R8 [28]. This requires no more than 9.4% of the area available per vault to implement PIM logic. The area required for each PIM accelerator depends on the PIM target being implemented,

[1]As a comparison, a typical system-on-chip (SoC) used in a consumer device has an area of 50–100 mm^2 [343, 353, 357, 373, 374].

and we report these area numbers in Sections 4.3–4.6.

4.3. Chrome Web Browser

A web browser is one of the most commonly-used applications by consumer device users [73], and is listed as one of the most common applications in many mobile benchmarks [142, 171, 289]. We study Google Chrome, which has claimed the majority of the mobile browsing market share for several years [277], and has over a billion active users [297].

The user perception of the browser speed is based on three factors: (1) page load time, (2) smooth web page scrolling, and (3) quick switching between browser tabs. In our study, we focus on two user interactions that impact these three factors, and govern a user's browsing experience: (1) page scrolling and (2) tab switching. Note that each interaction includes page loading.

4.3.1. Background

When a web page is downloaded, the rendering engine in Chrome, called Blink [64], parses the HTML to construct a Document Object Model (DOM) tree, which consists of the internal elements of the web page represented as tree nodes. Blink also parses the style rule data from the Cascading Style Sheet (CSS). The DOM tree and style rules enable a visual representation of the page, called the *render tree*. Each render tree node, called a *render object*, requires geometric information of the corresponding element to paint it to the display. The process of calculating the position and size of each render object is called *layout*. Once layout is complete, Chrome uses the Skia library [126] to perform *rasterization*, where a bitmap is generated for each render object by recursively traversing the render tree. The rasterized bitmap (also known as a *texture*) is then sent to the GPU through a process called *texture upload*, after which the GPU performs *compositing*. During compositing, the GPU paints the pixels corresponding to the texture onto the screen.

4.3.2. Page Scrolling

Scrolling triggers three operations: (1) *layout*, (2) *rasterization*, and (3) *compositing*. All three operations *must* happen within the mobile screen refresh time (60 FPS or 16.7 ms [243, 274]) to avoid frame dropping. Scrolling often forces the browser to recompute the layout for the new dimensions and position of each web page element, which is highly compute-intensive. However, a careful design of web pages can mitigate this cost [351, 359]. The browser also needs to rasterize any new objects that were not displayed previously. Depending on the web page contents and scroll depth, this requires significant computation [239, 359]. The updated rasterized bitmap must then be composited to display the results of the scroll on the screen.

4.3.2.1. Scrolling Energy Analysis

Figure 4.1 shows the energy breakdown for scrolling on different web pages. Across all of the pages that we test, a significant portion (41.9%) of page scrolling energy is spent on two data-intensive components: (1) *texture tiling*, where the graphics driver reorganizes the linear bitmap data into a tiled format for the GPU; and (2) *color blitting*, which is invoked by the Skia library [126] during rasterization. The rest of the energy is spent on a variety of other libraries and functions, each of which contributes to less than 1% of the total energy consumption (labeled *Other* in Figure 4.1).

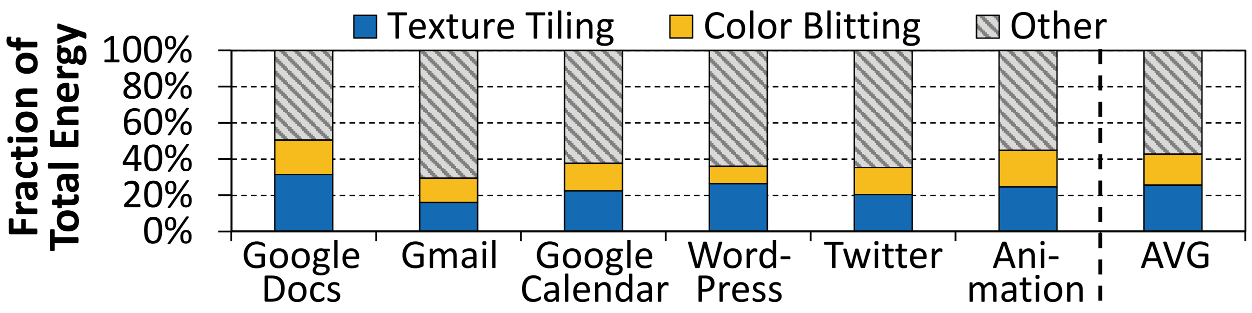


Figure 4.1. Energy breakdown for page scrolling.

Figure 4.2 (left) gives more insight into where energy is spent in the system when we scroll through a Google Docs [119] web page. We find that 77% of the total energy consumption is due to data movement. The data movement energy includes the energy consumed by DRAM, the off-chip interconnect, and the on-chip caches. The data movement generated by texture tiling and color blitting alone accounts for 37.7% of the total system energy (right graph in Figure 4.2). We confirm this by measuring the MPKI issued by the last-level cache (LLC). All of our pages exhibit a high MPKI (21.4 on average), and the majority of LLC misses are generated by texture tiling and color

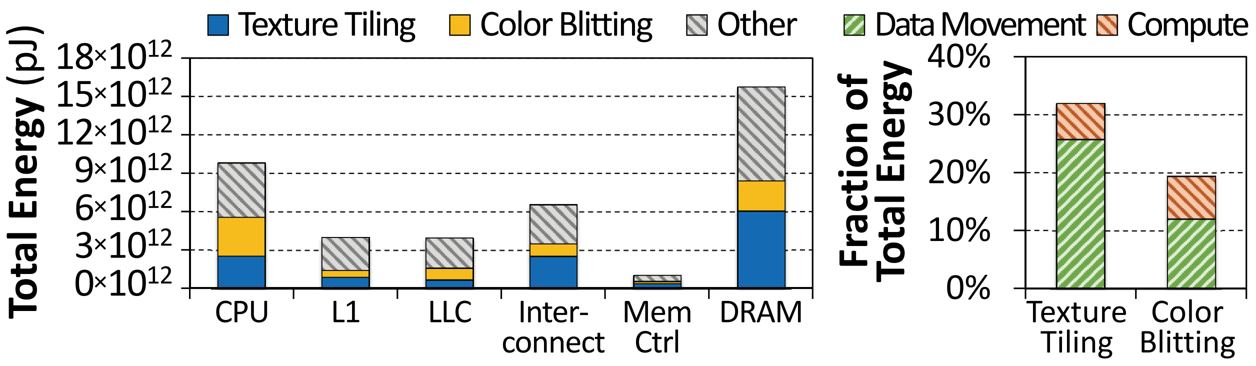


Figure 4.2. Energy breakdown when scrolling through a Google Docs web page.

blitting. Texture tiling and color blitting are also the top two contributors to the execution time, accounting for 27.1% of the cycles executed when scrolling through our Google Docs page.

We conclude that texture tiling and color blitting are responsible for a significant portion of the data movement that takes place during web page scrolling.

4.3.2.2. Analysis of PIM Effectiveness

In this section, we analyze the suitability of texture tiling and color blitting for PIM execution.

Texture Tiling. Figure 4.3a illustrates the major steps and associated data movement that take place during the texture tiling process. Texture tiling takes place after rasterization and before compositing. Rasterization generates a linear rasterized bitmap, which is written using a linear access pattern to memory (❶ in the figure). After rasterization, compositing accesses each texture in both the horizontal and vertical directions. To minimize cache misses during compositing, the graphics driver reads the rasterized bitmap (❷) and converts the bitmap into a *tiled* texture layout (❸). For example, the Intel HD Graphics driver breaks down each rasterized bitmap into multiple 4 kB texture tiles [173]. This allows the GPU to perform compositing on only one tile from the bitmap at a time to improve data locality.

As we observe from Figure 4.2 (right), 25.7% of the total system energy is spent on data movement generated by texture tiling. In fact, only 18.5% of the total energy consumed by texture tiling is used for computation, and the rest goes to data movement. The majority of the data movement comes from (1) the poor data locality during texture tiling; and (2) the large rasterized bitmap size (e.g., 1024x1024 pixels, which is 4 MB), which typically exceeds the LLC capacity.

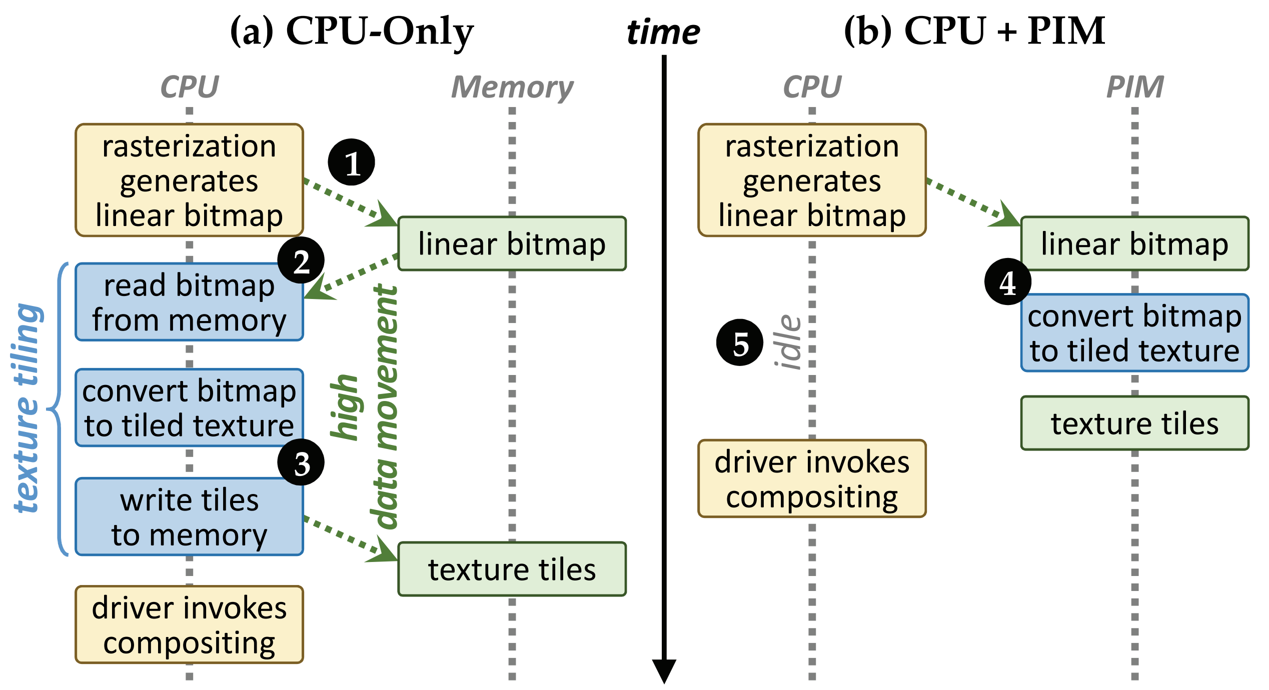

(a) CPU-Only
time
(b) CPU + PIM
CPU
Memory
rasterization generates linear bitmap
1
linear bitmap
2
read bitmap from memory
convert bitmap to tiled texture
3
write tiles to memory
texture tiling
high data movement
texture tiles
driver invokes compositing
CPU
PIM
rasterization generates linear bitmap
linear bitmap
4
convert bitmap to tiled texture
texture tiles
5
idle
driver invokes compositing

Figure 4.3. Texture tiling on (a) CPU vs. (b) PIM.

Due to the high amount of data movement, texture tiling is a good candidate for PIM execution. As shown in Figure 4.3b, by moving texture tiling to PIM (❹ in the figure), we can free up the CPU (❺) to perform other important compute-intensive tasks in Chrome, such as handling user interaction, executing JavaScript code, or rasterizing other render objects.

We next determine whether texture tiling can be implemented *in a cost-effective manner* using PIM. Our analysis indicates that texture tiling requires only simple primitives: *memcopy*, bitwise operations, and simple arithmetic operations (e.g., addition). These operations can be performed at high performance on our PIM core (see Section 4.2.3), and are amenable to be implemented as a fixed-function PIM accelerator. Our PIM accelerator for texture tiling consists of multiple *in-memory tiling units*. Each in-memory tiling unit consists of only a simple ALU, and operates on a single 4 kB tile. We empirically decide to use four in-memory tiling units in each PIM accelerator. Using the area estimation approach proposed in prior work [77], we estimate that the overhead of each PIM accelerator is less than 0.25 mm^2, which requires no more than 7.1% of the area available per vault for PIM logic (see Section 4.2.3). We conclude that the area required for texture tiling PIM logic is small, making the PIM logic feasible to implement in a consumer device with

3D-stacked memory. We evaluate the energy efficiency and performance of both the PIM core and PIM accelerator for texture tiling in Section 4.9.1.

An alternate way to reduce the data movement cost of texture tiling is to directly rasterize the content in the GPU, instead of the CPU, by using OpenGL primitives [66]. While GPU rasterization eliminates the need to move textures around the system, the GPU's highly-parallel architecture is not a good fit for rasterizing fonts and other small shapes [180]. We observe this issue in particular for text-intensive pages. We find that when Chrome uses the GPU to rasterize text-intensive pages, the page load time *increases* by up to 24.9%. This is one of the main reasons that the GPU rasterization is *not* enabled by default in Chrome. PIM can significantly reduce the overhead of texture tiling, while still allowing Chrome to exploit the benefits of CPU-based rasterization.

Color Blitting. During rasterization, the Skia library [126] uses high-level functions to draw basic primitives (e.g., lines, text) for each render object. Each of these high-level functions uses a *color blitter*, which converts the basic primitives into the bitmaps that are sent to the GPU. The primary operation of a blitter is to copy a block of pixels from one location to another. Blitting has many uses cases, such as drawing lines and filling paths, performing double buffering, alpha compositing, and combining two images or primitives together.

Color blitting requires simple computation operations, but generates a large amount of data movement. As we see in Figure 4.2 (right), color blitting accounts for 19.1% of the total system energy used during page scrolling. 63.9% of the energy consumed by color blitting is due to data movement (right graph in Figure 4.2), primarily due to its streaming access pattern and the large sizes of the bitmaps (e.g., 1024x1024 pixels). Similar to texture tiling, color blitting is a good candidate for PIM execution due to its high amount of data movement.

We next determine whether color blitting can be implemented in a cost-effective manner using PIM. Our analysis reveals that color blitting requires only low-cost computations such as *memset*, simple arithmetic operations to perform alpha blending (e.g., addition and multiplication), and shift operations. These operations can be performed at high performance on our PIM core (see Section 4.2.3), or we can use a PIM accelerator that consists of the same four in-memory logic units that we design for texture tiling, with different control logic specifically designed to perform color blitting. Thus, we conclude that the area required for color blitting PIM logic is small, and that the PIM logic is feasible to implement in a consumer device with 3D-stacked memory. We evaluate the energy efficiency and performance of both the PIM core and PIM accelerator for color blitting in Section 4.9.1.

4.3.3. Tab Switching

Chrome uses a multi-process architecture, where each tab has its own application process for rendering the page displayed in the tab (see Section 4.3.1), to enhance reliability and security. When a user switches between browser tabs, this triggers two operations: (1) a context switch to the process for the selected tab, and (2) a load operation for the new page. There are two primary concerns during tab switching: (1) how fast a new tab loads and becomes interactive, as this directly affects user satisfaction; and (2) memory consumption. Memory consumption is a major concern for three reasons. First, the average memory footprint of a web page has increased significantly in recent years [170] due to the increased use of images, JavaScript, and video in modern web pages. Second, users tend to open multiple tabs at a time [79, 281], with each tab consuming additional memory. Third, consumer devices typically have a much lower memory capacity than server or desktop systems, and without careful memory management, Chrome could run out of memory on these systems over time.

One potential solution to handle a memory shortage is to kill inactive background tabs, and when the user accesses those tabs again, reload the tab pages from the disk. There are two downsides to this mechanism. First, reloading tabs from the disk (which invokes page faults) and rebuilding the page objects take a relatively long time. Second, such reloading may lead to the loss of some parts of the page (e.g., active remote connections). To avoid these drawbacks, Chrome uses memory compression to reduce each tab's memory footprint. When the available memory is lower than a predetermined threshold, Chrome compresses pages of an inactive tab, with assistance from the OS, and places them into a DRAM-based memory pool, called ZRAM [190]. When the user switches to a previously-inactive tab whose pages were compressed, Chrome loads the data from ZRAM and decompresses it. This allows the browser to avoid expensive I/O operations to disk, and retrieve the web page data much faster, thereby improving overall performance and the browsing experience.

4.3.3.1. Tab Switching Energy Analysis

To study data movement during tab switching, we perform an experiment where a user (1) opens 50 tabs (picked from the top most-accessed websites [19]), (2) scrolls through each tab for a few seconds, and then (3) switches to the next tab. During this study, we monitor the amount of data that is swapped in and out of ZRAM per second, which we plot in Figure 4.4. We observe that, in total, 11.7 GB of data is swapped out to ZRAM (left graph in Figure 4.4), at a rate of up to 201 MB/s. When the CPU performs this swapping, as shown in Figure 4.5a, it incurs significant overhead, as it must read the inactive page data (❶ in the figure), compress the data (❷), and write the compressed data to ZRAM (❸). We observe a similar behavior for decompression (right graph in Figure 4.4), with as much as 7.8 GB of data swapped in, at a rate of up to 227 MB/s. Note that during decompression, most of the newly-decompressed page cache lines are not accessed by the CPU, as the tab rendering process needs to read only a small fraction of the decompressed cache lines to display the tab's contents. When switching between 50 tabs, compression and decompression incur 19.6 GB of data movement, and contribute to 18.1% and 14.2% of the total system energy and execution time of tab switching, respectively.

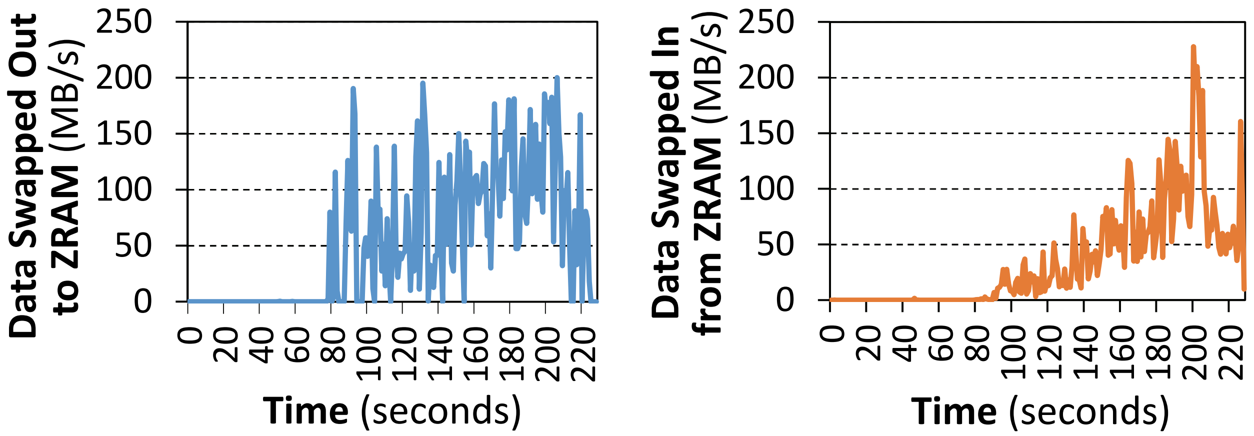


Figure 4.4. Number of bytes per second swapped out to ZRAM (left) and in from ZRAM (right), while switching between 50 tabs.

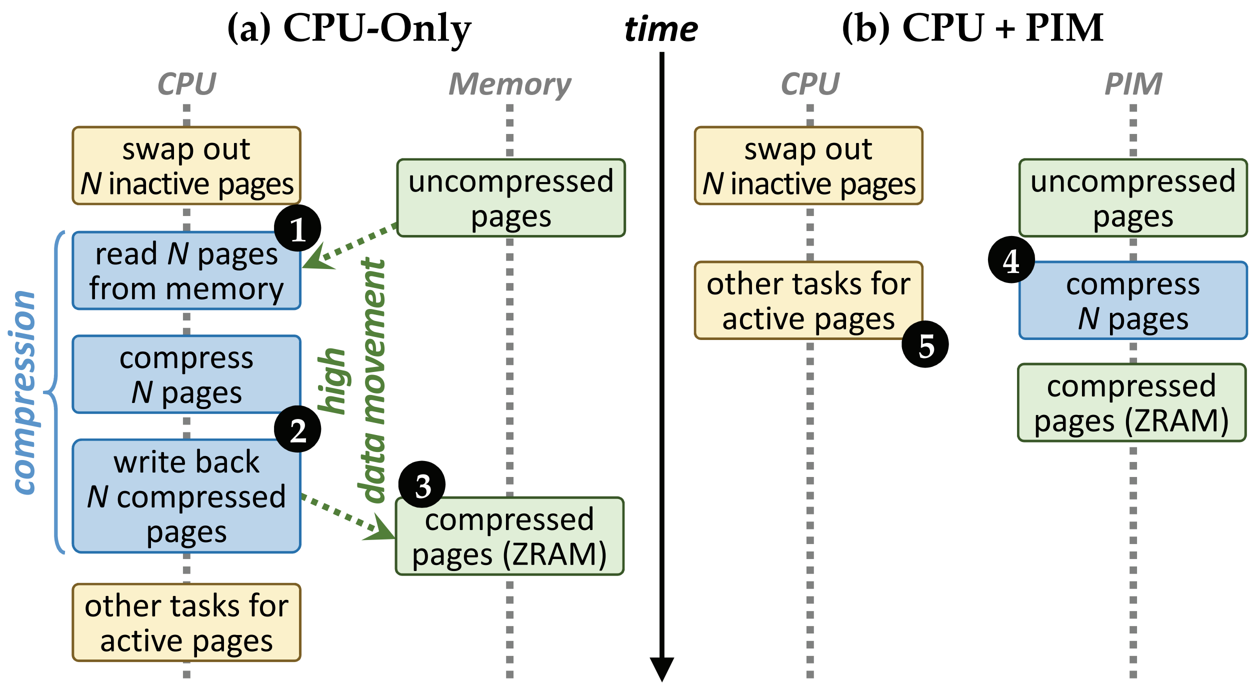


Figure 4.5. Compression on (a) CPU vs. (b) PIM.

4.3.3.2. Analysis of PIM Effectiveness

Compression and decompression are a good fit for PIM execution, as (1) they cause a large amount of data movement; and (2) compression can be handled in the background, since none of the active tab processes need the data that is being compressed. Figure 4.5b shows the compression operation once it has been offloaded to PIM. In this case, the CPU only informs the PIM logic about the pages that need to be swapped out, and the PIM logic operates on the uncompressed page data that is already in memory (❹ in the figure). This (1) eliminates the off-chip page data movement that took place during CPU-based compression, and (2) frees up the CPU to perform other tasks while PIM performs compression in the background (❺). We observe a similar behavior for tab decompression, where the compressed data is kept within DRAM, and only those cache lines used later by the CPU are sent across the off-chip channel, reducing data movement and improving CPU cache utilization.

We find that compression and decompression are good candidates for PIM execution. Chrome's ZRAM uses the LZO compression algorithm [283] which uses simple operations and favors speed over compression ratio [194, 283]. Thus, LZO can execute without performance loss on our PIM core (see Section 4.2.3). Prior work [393] shows that sophisticated compression algorithms (e.g., LZ77 compression [401] with Huffman encoding [172]) can be implemented efficiently using an accelerator. Thus, we expect that the simpler LZO compression algorithm can be implemented as a PIM accelerator that requires less than 0.25 mm^2 of area [393]. Thus, both the PIM core and PIM accelerator are feasible to implement in-memory compression/decompression.

In-memory compression/decompression can benefit a number of other applications and use cases. For example, many modern operating systems support user-transparent file system compression (e.g., BTRFS [311] and ZFS [41]). Such compression is not yet widely supported in commercial *mobile* operating systems due to energy and performance issues [393]. An in-memory compression unit can enable efficient user-transparent file system compression by eliminating the off-chip data movement cost and largely reducing the decompression latency.

4.4. TensorFlow Mobile

Machine learning (ML) is emerging as an important core function for consumer devices. Recent works from academia and industry are pushing inference to mobile devices [22, 49, 58, 303, 307], as opposed to performing inference on cloud servers. In this work, we study TensorFlow Mobile [128], a version of Google's TensorFlow ML library that is specifically tailored for mobile and embedded platforms. TensorFlow Mobile enables a variety of tasks, such as image classification, face recognition, and Google Translate's instant visual translation [125], all of which perform inference on consumer devices using a neural network that was pre-trained on cloud servers.

4.4.1. Background

Inference begins by feeding input data (e.g., an image) to a neural network. A neural network is a directed acyclic graph consisting of multiple layers. Each layer performs a number of calculations and forwards the results to the next layer. Depending on the type of the layer, the calculation can differ for each level. A fully-connected layer performs matrix multiplication (MatMul) on the input data, to extract high-level features. A 2-D convolution layer applies a convolution filter (Conv2D) across the input data, to extract low-level features. The last layer of a neural network is the output layer, which performs classification to generate a prediction based on the input data.

4.4.2. TensorFlow Mobile Energy Analysis

Figure 4.6 shows the breakdown of the energy consumed by each function in TensorFlow Mobile, for four different input networks. As convolutional neural networks (CNNs) consist mainly of 2-D convolution layers and fully-connected layers [11], the majority of energy is spent on these two types of layers. However, we find that there are two other functions that consume a significant fraction of the system energy: *packing/unpacking* and *quantization*. Packing and unpacking reorder the elements of matrices to minimize cache misses during matrix multiplication. Quantization converts 32-bit floating point and integer values into 8-bit integers to improve the execution time and energy consumption of inference. These two together account for 39.3% of total system energy on average. The rest of the energy is spent on a variety of other functions such as random sampling, reductions, and simple arithmetic, and each of which contributes to less than 1% of total energy consumption (labeled *Other* in Figure 4.6).

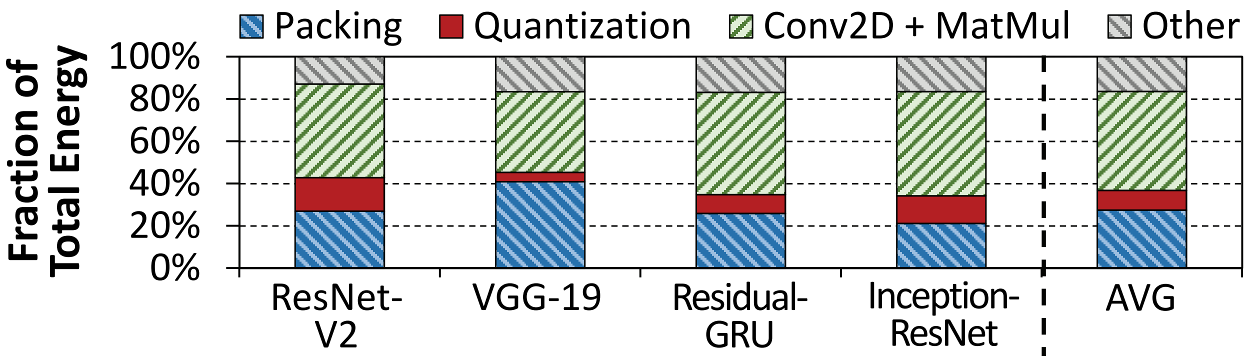


Figure 4.6. Energy breakdown during inference execution on four input networks.

Further analysis (not shown) reveals that data movement between the CPUs and main memory accounts for the majority of total system energy consumed by TensorFlow Mobile. While Conv2D and MatMul dominate CPU energy consumption, 57.3% of the total system energy is spent on data movement, on average across our four input networks. We find that 54.4% of the data movement energy comes from packing/unpacking and quantization.

Even though the main goal of packing and quantization is to reduce energy consumption and inference latency, our analysis shows that they generate a large amount of data movement, and thus, lose part of the energy savings they aim to achieve. Furthermore, Figure 4.7 shows a significant portion (27.4% on average) of the execution time is spent on the packing and quantization process. Hence, we focus our analysis on these two functions. We exclude Conv2D and MatMul from our analysis because (1) a majority (67.5%) of their energy is spent on computation; and (2) Conv2D and MatMul require a relatively large and sophisticated amount of PIM logic [99, 206], which may not be cost-effective for consumer devices.

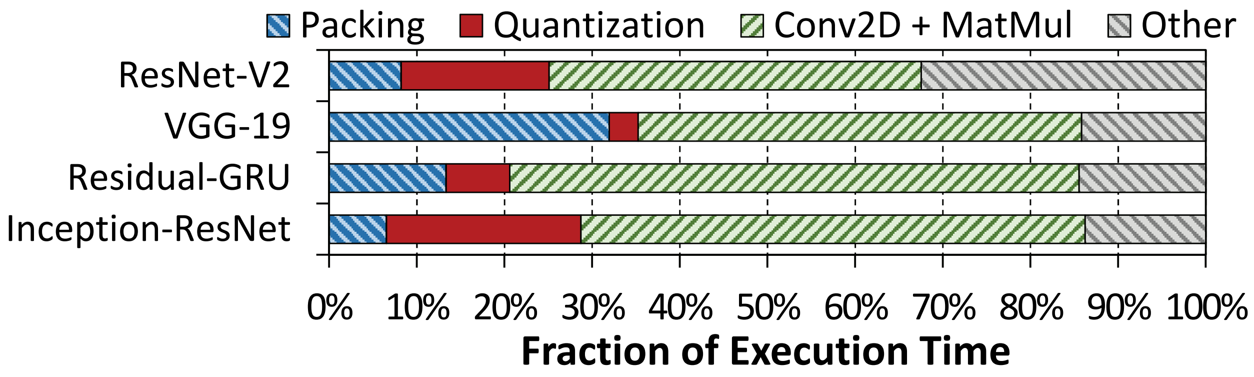


Figure 4.7. Execution time breakdown of inference.

4.4.3. Analysis of PIM Effectiveness

Packing. GEneralized Matrix Multiplication (GEMM) is the core building block of neural networks, and is used by both 2-D convolution and fully-connected layers. These two layers account for the majority of TensorFlow Mobile execution time. To implement fast and energy-efficient GEMM, TensorFlow Mobile employs a low-precision, quantized GEMM library called *gemmlowp* [116]. The gemmlowp library performs GEMM by executing its innermost kernel, an architecture-specific GEMM code portion for small fixed-size matrix chunks, multiple times. First, gemmlowp fetches matrix chunks which fit into the LLC from DRAM. Then, it executes the GEMM kernel on the fetched matrix chunks in a block-wise manner.

To minimize cache misses, gemmlowp employs a process called *packing*, which reorders the matrix chunks based on the memory access pattern of the kernel to make the chunks cache-friendly. After performing GEMM, gemmlowp performs *unpacking*, which converts the result matrix chunk back to its original order.

Packing and unpacking account for up to 40% of the total system energy and 31% of the inference execution time, as shown in Figures 4.6 and 4.7, respectively. Due to their unfriendly cache access pattern and the large matrix sizes, packing and unpacking generate a significant amount of data movement. For instance, for VGG-19 [339], 35.3% of the total energy goes to data movement incurred by packing-related functions. On average, we find that data movement is responsible for 82.1% of the total energy consumed during the packing/unpacking process, indicating that packing and unpacking are bottlenecked by data movement.

Packing and unpacking are simply pre-processing steps, to prepare data in the right format for the kernel. Ideally, the CPU should execute only the GEMM kernel, and assume that packing and unpacking are already taken care of. PIM can enable such a scenario by performing packing and unpacking without *any* CPU involvement. Our PIM logic packs matrix chunks, and sends the packed chunks to the CPU, which executes the GEMM kernel. Once the GEMM kernel completes, the PIM logic receives the result matrix chunk from the CPU, and unpacks the chunk while the CPU executes the GEMM kernel on a different matrix chunk.

We next determine whether packing and unpacking can be implemented in a cost-effective manner using PIM. Our analysis reveals that packing is a simple data reorganization process and requires simple arithmetic operations to scan over matrices and compute new indices. As a result, packing and unpacking can be performed at high performance on our PIM core (see Section 4.2.3), or on a PIM accelerator that consists of the same four in-memory logic units that we design for texture tiling (see Section 4.3.2.2), with different control logic specifically designed to perform packing and unpacking. For packing and unpacking, we can assign each in-memory logic unit to work on one chunk of the matrix. Thus, we conclude that it is feasible to implement packing and unpacking using PIM in a consumer device with 3D-stacked memory.

Quantization. TensorFlow Mobile performs quantization twice for each Conv2D operation. First, quantization is performed on the 32-bit input matrix before Conv2D starts. Then, Conv2D runs, during which gemmlowp generates a 32-bit result matrix.[2] Quantization is performed for the second

[2]Even though gemmlowp reads 8-bit input matrices, its output matrix uses 32-bit integers, because multiplying two 8-bit integers produces a 16-bit result, and gemmlowp uses a 32-bit integer to store each 16-bit result.

time on this result matrix (this step is referred to as *re-quantization*). Accordingly, invoking Conv2D more frequently (which occurs when there are more 2-D convolution layers in a network) leads to higher quantization overheads. For example, VGG requires only 19 Conv2D operations, incurring small quantization overheads. On the other hand, ResNet requires 156 Conv2D operations, causing quantization to consume 16.1% of the total system energy and 16.8% of the execution time. The quantization overheads are expected to increase as neural networks get deeper.

Figure 4.8a shows how TensorFlow quantizes the result matrix using the CPU. First, the entire matrix needs to be scanned to identify the minimum and maximum values of the matrix (❶ in the figure). Then, using the minimum and maximum values, the matrix is scanned a second time to convert each 32-bit element of the matrix into an 8-bit integer (❷). These steps are repeated for re-quantization of the result matrix (❸ and ❹). The majority of the quantization overhead comes from data movement. Because both the input matrix quantization and the result matrix re-quantization need to scan a large matrix twice, they exhibit poor cache locality and incur a large amount of data movement. For example, for the ResNet network, 73.5% of the energy consumed during quantization is spent on data movement, indicating that the computation is relatively cheap (in comparison, only 32.5% of Conv2D/MatMul energy goes to data movement, while the majority goes to MAC computation). 19.8% of the total data movement energy of inference execution comes from quantization. As Figure 4.8b shows, we can offload both quantization (❺ in the figure) and re-quantization (❻) to PIM to eliminate data movement. This frees up the CPU to focus on GEMM execution, and allows the next Conv2D operation to be performed in parallel with re-quantization (❼).

Quantization itself is a simple data conversion operation that requires shift, addition, and multiplication operations. As a result, quantization can be performed at high performance on our PIM core (see Section 4.2.3), or on a PIM accelerator that consists of the same four in-memory logic units that we design for texture tiling (see Section 4.3.2.2), with different control logic specifically designed to perform quantization. For quantization, we can assign each in-memory logic unit to work on a separate Conv2D operation. Thus, we conclude that it is feasible to implement quantization using PIM in a consumer device with 3D-stacked memory.

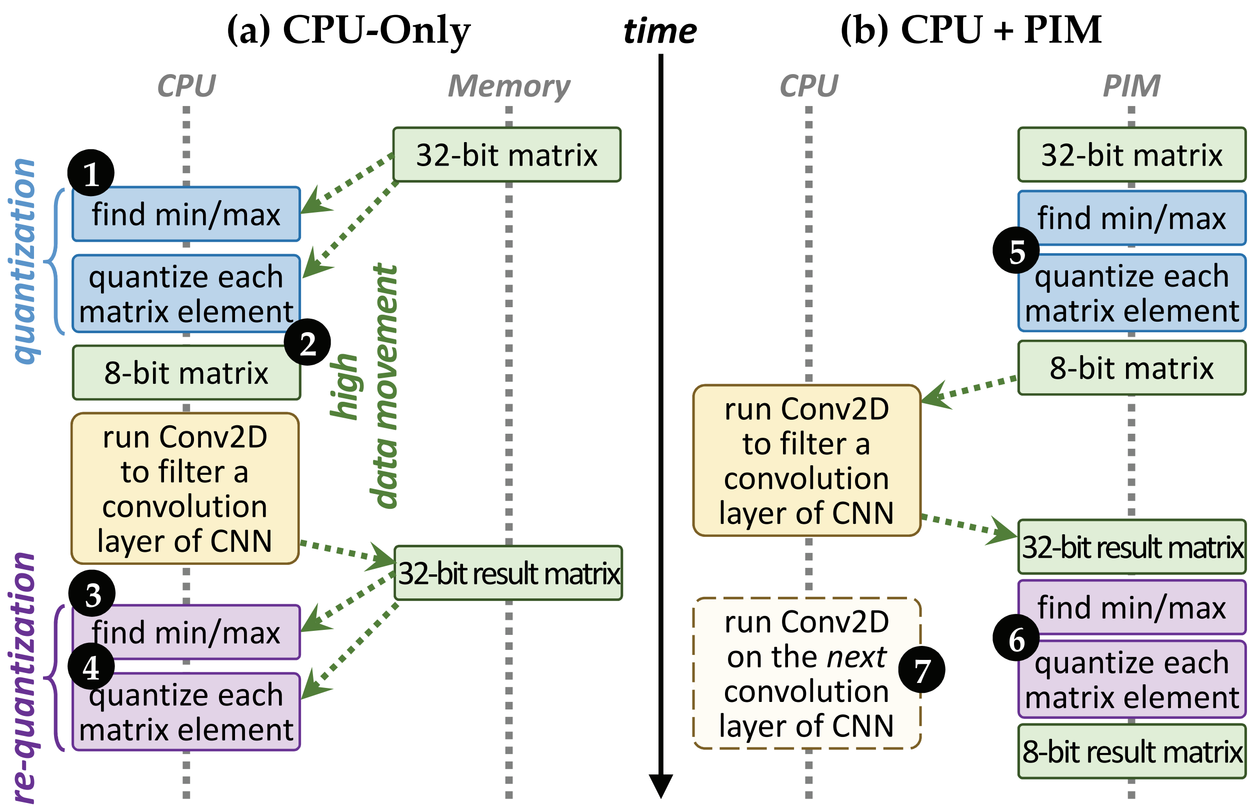

(a) CPU-Only
time
(b) CPU + PIM
CPU
Memory
CPU
PIM
quantization
re-quantization
high data movement
32-bit matrix
1
find min/max
quantize each matrix element
2
8-bit matrix
run Conv2D to filter a convolution layer of CNN
32-bit result matrix
3
find min/max
4
quantize each matrix element
32-bit matrix
find min/max
5
quantize each matrix element
8-bit matrix
run Conv2D to filter a convolution layer of CNN
32-bit result matrix
find min/max
6
quantize each matrix element
8-bit result matrix
run Conv2D on the next convolution layer of CNN
7

Figure 4.8. Quantization on (a) CPU vs. (b) PIM.

4.5. Video Playback

Video playback is among the most heavily- and commonly-used applications among mobile users [390]. Recent reports on mobile traffic [68, 85] show that video dominates consumer device traffic today. Many Google services and products, such as YouTube [130], Google Hangouts [120], and the Chrome web browser [112] rely heavily on video playback. YouTube alone has over a billion users, and the majority of video views on YouTube come from mobile devices [131]. Video playback requires a dedicated decoder, which decompresses and decodes the streaming video data and renders video on the consumer device. In this work, we focus on the VP9 decoder [134], which is an open-source codec widely used by video-based applications, and is supported by most consumer devices [370].

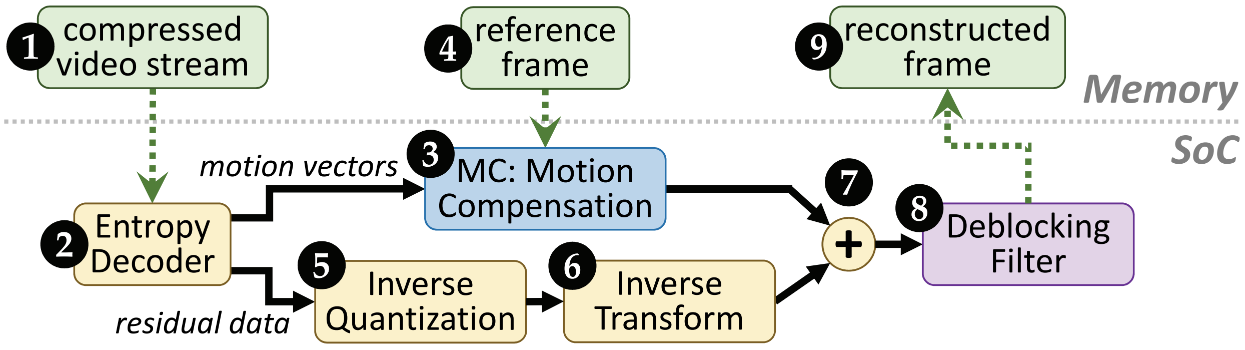


Figure 4.9. General overview of the VP9 decoder.

4.5.1. Background

Figure 4.9 shows a high-level overview of the VP9 decoder architecture. VP9 processes video one frame at a time, and uses a *reference frame* to decode the current frame. For the current frame, the decoder reads the frame's compressed content (i.e., the input bitstream) from memory (❶ in the figure) and sends the content to the *entropy decoder*. The entropy decoder (❷) extracts (1) *motion vectors*, which predict how objects move relative to the reference frame; and (2) *residual data*, which fills in information not predicted by the motion vectors.

The motion vectors are sent to the *motion compensation* (MC) unit (❸), which applies the object movement predictions to the reference frame (❹). MC predicts movement at a macro-block (i.e., 16x16-pixel) granularity. Each frame is decomposed into macro-blocks, and each motion vector points to a macro-block in the reference frame. VP9 allows motion vectors to have a resolution as small as $\frac{1}{8}$th of a pixel, which means that a motion vector may point to a pixel location with a non-integer value. In such cases, MC performs *sub-pixel interpolation* to estimate the image at a non-integer pixel location. In parallel with MC, the residual data is transformed by the *inverse quantization* (❺) and *inverse transform* (❻) blocks. The resulting information is combined with the macro-block output by MC (❼) to reconstruct the current frame.

Due to block-based prediction, there may be discontinuities at the border between two blocks. The *deblocking filter* (❽) attempts to remove such artifacts by (1) identifying *edge pixels* (i.e., pixels that lie on the border of two blocks) that are discontinuous with their neighbors, and (2) applying a low-pass filter to these pixels. Finally, the reconstructed frame (❾) is written back to the frame buffer in memory.

The VP9 decoder architecture can be implemented in either software or hardware. We provide an extensive analysis of two decoder implementations in this section: (1) libvpx [371], Google's open-source software decoder library, which is used as a reference software implementation for the VP9 codec; and (2) Google's VP9 hardware decoder [372].

4.5.2. VP9 Software Decoder

4.5.2.1. Software Decoder Energy Analysis

Figure 4.10 shows the total system energy consumed when libvpx VP9 decoder is used to play a 4K (3840x2160-pixel) video from Netflix [382]. We find that the majority of energy (53.4%) is spent on MC, which is the most bandwidth-intensive and time-consuming task in the decoder [45, 164]. Prior works [143, 229, 308] make similar observations for other video codecs. Most of the energy consumption in MC is due to sub-pixel interpolation, the most memory-intensive component of MC [45, 164, 192]. We find that sub-pixel interpolation alone consumes 37.5% of the total energy and 41.2% of the execution time spent *by the entire decoder*. Aside from MC, the deblocking filter is another major component of the decoder, consuming 29.7% of the software decoder energy and 24.3% of the total cycle count. Other components, such as the entropy decoder and inverse transform, consume a smaller portion of the energy.

Figure 4.11 shows the per-function breakdown of the energy consumption of each hardware component when the VP9 software decoder is used to play back 4K video. We find that 63.5% of the total energy is spent on data movement, the majority (80.4%) of which is performed by MC

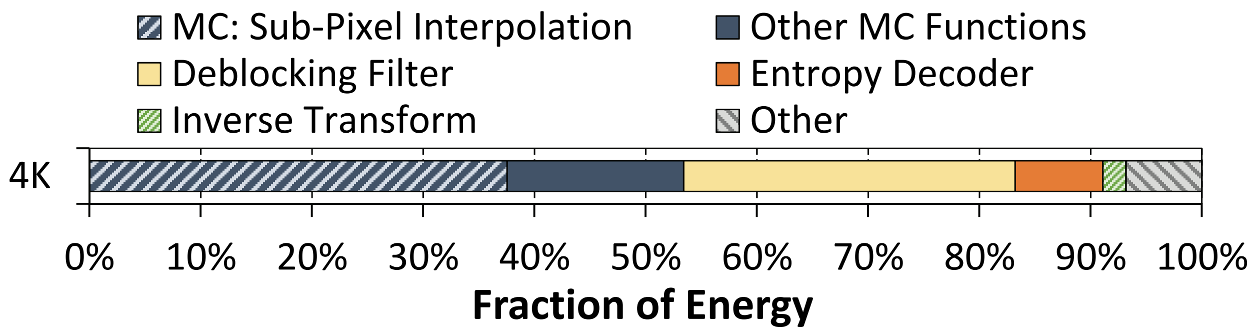


Figure 4.10. Energy analysis of VP9 software decoder.

and the deblocking filter. Sub-pixel interpolation is the dominant function of MC, and on its own generates 42.6% of the *total* data movement. We find that the CPU spends the majority of its time and energy stalling (not shown in Figure 4.11) as it waits for data from memory during MC and deblocking filter execution. The other functions generate much less data movement because their working set fits within the CPU caches. For example, the entropy decoder works on the encoded bit stream and the inverse transform works on the decoded coefficients, and both are small enough to be captured by the caches.

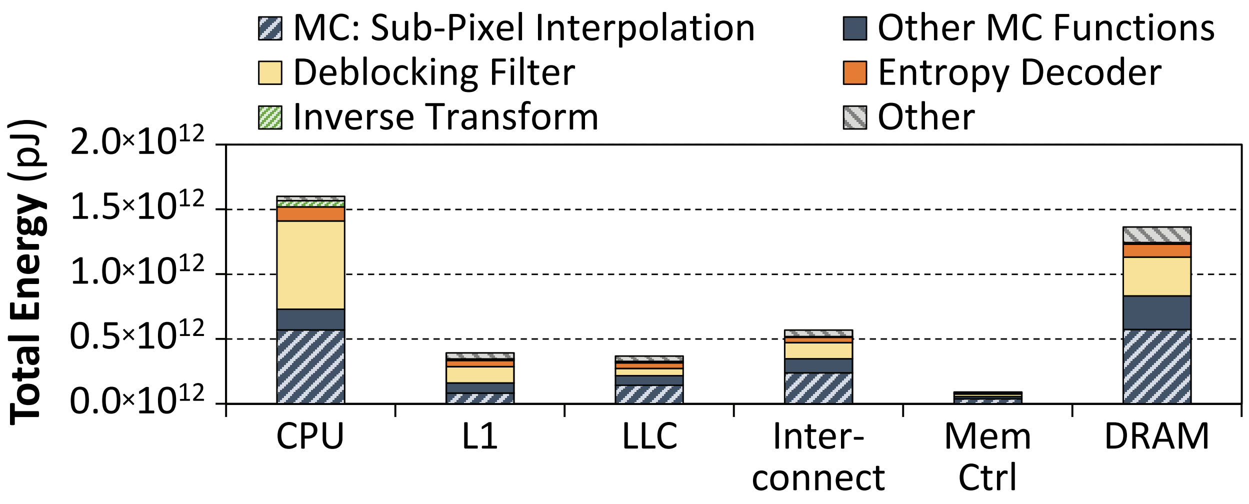


Figure 4.11. Energy breakdown of VP9 software decoder.

We conclude that video playback generates a large amount of data movement, the majority of which comes from sub-pixel interpolation and the deblocking filter.

4.5.2.2. Analysis of PIM Effectiveness

Sub-Pixel Interpolation. During sub-pixel interpolation, VP9 uses a combination of multi-tap finite infinite response (FIR) and bi-linear filters to interpolate the value of pixels at non-integer locations [134]. The interpolation process is both compute- and memory-intensive, for two reasons. First, interpolating each sub-pixel value requires multiple pixels to be fetched from memory. VP9 decomposes each frame into 64x64-pixel superblocks, which are then divided further into smaller sub-blocks, as small as 4x4-pixel. In the worst case, where the decoder interpolates pixels in a sub-block at a $\frac{1}{8}$-pixel resolution, the decoder fetches 11x11 pixels from the reference frame.

Second, each motion vector can point to *any* location in the reference frame. As a result, there is poor locality, and sub-pixel interpolation does *not* benefit significantly from caching. We confirm this in Figure 4.11, where we observe that data movement between main memory and the CPU accounts for 65.3% of the total energy consumed by sub-pixel interpolation. As a result, sub-pixel interpolation is a good candidate for PIM execution.

We find that sub-pixel interpolation is amenable for PIM, as it consists mainly of multiplication, addition, and shift operations [134, 164, 192]. As a result, sub-pixel interpolation can be executed with high performance on our PIM core (see Section 4.2.3), and can be implemented as a fixed-function PIM accelerator. Our PIM accelerator for sub-pixel interpolation is similar in design to the sub-pixel interpolation component of the hardware VP9 decoder. Our analysis shows that our PIM accelerator occupies an area of 0.21 mm^2, which requires no more than 6.0% of the area available per vault for PIM logic (see Section 4.2.3). Thus, we conclude that it is feasible to implement sub-pixel interpolation using PIM in a consumer device with 3D-stacked memory.

Deblocking Filter. Recall that the deblocking filter works on the edge pixels in each 64x64-pixel superblock, to remove blocking artifacts. Deblocking invokes a low-pass filter on neighboring edge pixels, and iterates through the superblocks in a raster scan order [134]. For each edge between two superblocks, the filter evaluates up to eight pixels on either side of the edge, and if the filter condition is triggered, the filter may modify up to seven pixels on either side.

Like sub-pixel interpolation, the deblocking filter is compute- and memory-intensive, and accounts for a third of the computation complexity of the entire VP9 decoder [61, 70]. As the filter needs to check the vertical and horizontal edges of each 4x4-pixel block in the frame, its memory access pattern exhibits poor cache locality. Instead, the filter generates a large amount of data movement and produces strictly less output than input, with 71.1% of the movement taking place on the off-chip memory channel. Hence, the deblocking filter is a good fit for PIM.

We find that the deblocking filter is amenable for PIM, as it is a simple low-pass filter that requires only arithmetic and bitwise operations. As a result, the deblocking filter can be executed with high performance on our PIM core (see Section 4.2.3), and can be implemented as a fixed-function PIM accelerator that is is similar in design to the deblocking filter component of the hardware VP9 decoder. Our analysis shows that our PIM accelerator occupies an area of $0.12\,\text{mm}^2$, which requires no more than 3.4% of the area available per vault for PIM logic (see Section 4.2.3). Thus, we conclude that it is feasible to implement the deblocking filter using PIM in a consumer device with 3D-stacked memory.

4.5.3. VP9 Hardware Decoder

For high-resolution (e.g., 4K) video playback, a hardware decoder is essential. It enables fast video decoding at much lower energy consumption than a software decoder. In this section, we present our analysis on the VP9 hardware decoder, which is used in a large fraction of consumer devices [370].

Unlike the software decoder, the VP9 hardware decoder employs several techniques to hide memory latency, such as prefetching and parallelization. For example, the hardware decoder can work on a batch of motion vectors simultaneously, which allows the decoder to fetch the reference pixels for the entire batch at once. In addition, the hardware decoder does not make use of the CPU caches, and instead sends requests directly to DRAM, which allows the decoder to avoid cache lookup latencies and cache coherence overheads. Thus, there is little room to improve the memory latency for the hardware decoder. However, we find that despite these optimizations, there is still a significant amount of off-chip data movement generated by the hardware decoder, which consumes significant memory bandwidth and system energy.

4.5.3.1. Hardware Decoder Energy Analysis

Figure 4.12 shows the breakdown of off-chip traffic when decoding one frame of a 4K video and an HD (720x1280-pixel) video. For each resolution, we show the traffic both with and without lossless frame compression. We make five key observations from the analysis. First, the majority of the traffic (up to 75.5% for HD and 59.6% for 4K) comes from reading the reference frame data from DRAM during MC. This data movement is responsible for 71.3% and 69.2% of total hardware decoder energy, respectively. Note that while frame compression reduces the reference frame traffic, reference frame data movement still accounts for a significant portion (62.2% for HD and 48.8%

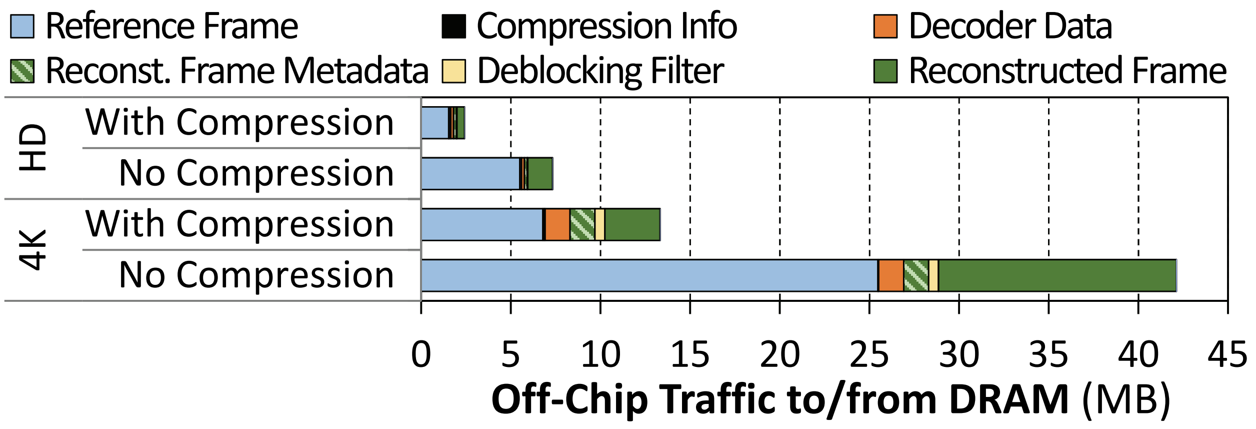


Figure 4.12. Off-chip traffic breakdown of VP9 hardware decoder.

for 4K) of the off-chip traffic. Second, similar to our observations for the software decoder, the bulk of this data movement is due to the extra reference pixels required by sub-pixel interpolation. For every pixel of the current frame, the decoder reads 2.9 reference frame pixels from DRAM. Third, data movement increases significantly as the video resolution increases. Our analysis indicates that decoding one 4K frame requires 4.6x the data movement of a single HD frame. Fourth, the reconstructed frame data output by the decoder is the second-biggest contributor to the off-chip data movement, generating 22.2% of the total traffic. Fifth, unlike the software decoder, the deblocking filter does not generate much off-chip data movement in the hardware decoder, as the hardware employs large SRAM buffers (875 kB) to cache the reference pixels read during MC, so that the pixels can be reused during deblocking.

4.5.3.2. Analysis of PIM Effectiveness

Similar to our observations for the software decoder (see Section 4.5.2), the MC unit (sub-pixel interpolation, in particular) is responsible for the majority of off-chip data movement in the hardware VP9 decoder. The MC unit is a good fit for PIM execution, because we can eliminate the need to move reference frames from memory to the on-chip decoder, thus saving significant energy. Figure 4.13 shows a high-level overview of our proposed architecture, and how a modified hardware decoder interacts with memory and with the *in-memory* MC unit. In this architecture, the entropy decoder (❶ in the figure) sends the motion vectors to memory, where the MC unit now resides. The in-memory MC unit (❷) generates macro-blocks using the motion vectors and the reference frame

(❸) fetched from memory. The macro-blocks generated by MC are then combined with residual data and sent to the deblocking filter (❹). To avoid sending the macro-blocks back to the decoder (which would eliminate part of the energy savings of not moving reference frame data), we also made the design choice of moving the deblocking filter into memory (❹). While the deblocking filter itself does *not* generate significant traffic, by placing it in PIM, we can avoid moving the reconstructed frame (❺) back and forth on the main memory bus unnecessarily.

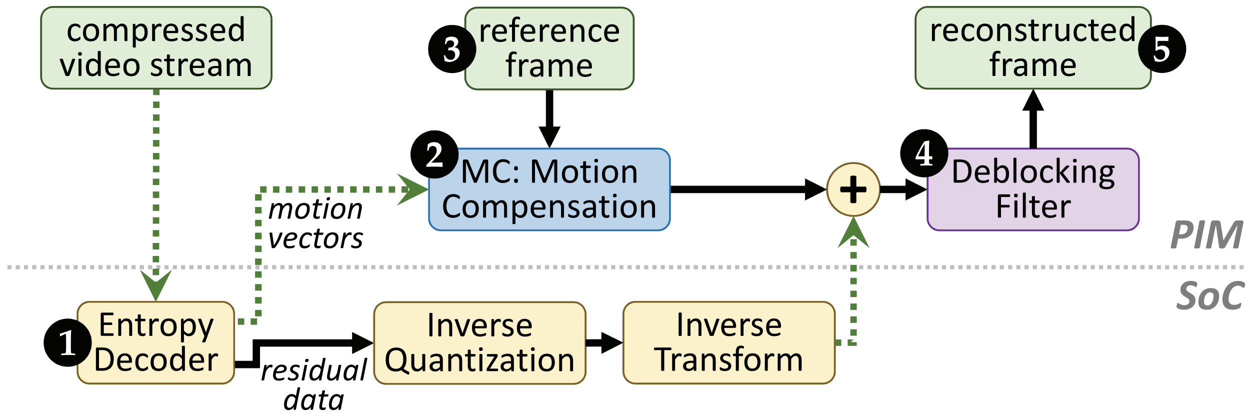


Figure 4.13. Modified VP9 decoder with in-memory MC.

Overall, moving the MC unit and deblocking filter reduces data movement significantly, as only the compressed input bitstream, the motion vectors, and the residual data need to move between the decoder hardware components and the memory. Another benefit of this approach is that we can use the same in-memory components for software VP9 decoding as well. In total, the area used by a PIM accelerator that performs MC and the deblocking filter is 0.33 mm^2, which requires no more than 9.4% of the area available per vault for PIM logic (see Section 4.2.3). We conclude that it is effective to perform part of VP9 hardware decoding in memory.

4.6. Video Capture

Video capture is used in many consumer applications, such as video conferencing (e.g., Google Hangouts [120], Skype [259]), video streaming (e.g., YouTube [130], Instagram [88]), and recording local video from a mobile device camera. Video-capture-based applications account for a significant portion of mobile traffic [68, 85, 390]. It is expected that video will constitute 78% of all mobile data traffic by 2021 [68]. To effectively capture, store, and transfer video content over the network, a video encoder is required. The video encoder handles the majority of the computation for video capture. In this work, we focus on the VP9 encoder, as it is the main compression technology used by Google Hangouts [120], YouTube [130], and most web-based video applications in the Chrome browser [112], and is supported by many mobile devices [370].

4.6.1. Background

Figure 4.14 shows a high-level overview of the VP9 encoder architecture. The encoder fetches the current frame (❶ in the figure) and three reference frames (❷) from DRAM. Then, the current frame is sliced into 16x16-pixel macro-blocks. Each macro-block is passed into the *intra-prediction* (❸) and *motion estimation* (ME) (❹) units. Intra-prediction predicts the value of an entire macro-block by measuring its similarity to adjacent blocks within the frame. ME, also known as *inter-prediction*, compresses the frame by exploiting temporal redundancy between adjacent frames. Then, ME finds the motion vector, which encodes the spatial offset of each macro-block relative to the reference frames. If successful, ME replaces the macro-block with the motion vector. The *mode decision unit* (❺) examines the outputs of intra-prediction and ME for each macro-block, and chooses the best prediction for the block. The macro-block prediction is then (1) transformed into the frequency domain using a discrete cosine (DCT) *transform* (❻), and (2) simultaneously sent to the motion compensation (MC) unit (❼) to reconstruct the currently-encoded frame, which will be used as a reference frame (❽) to encode *future* frames. The encoder then uses quantization (❾) to compress the DCT coefficients of each macro-block. Finally, the *entropy coder* step (❿) uses *run-length coding* and *variable-length coding* to reduce the number of bits required to represent the DCT coefficients.

As is the case for the VP9 decoder, the encoder can be implemented in either software or hardware. We analyze both libvpx [371], Google's open source software encoder library, and Google's VP9 hardware encoder [372].

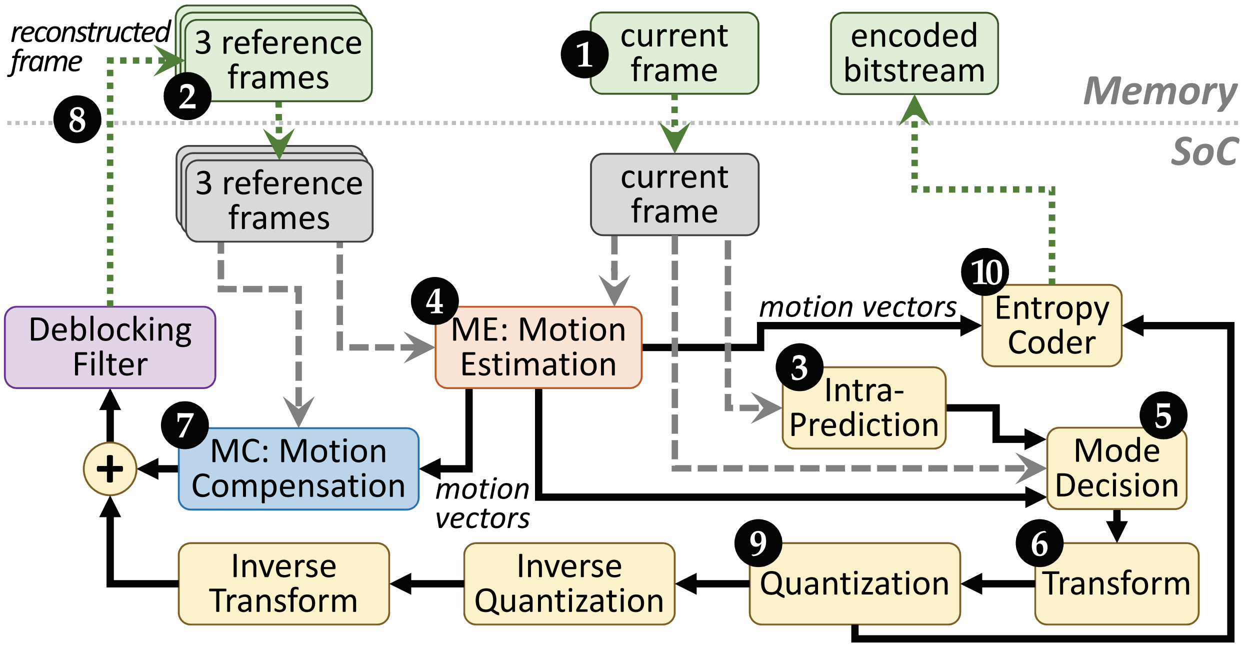


Figure 4.14. General overview of the VP9 encoder.

4.6.2. VP9 Software Encoder

4.6.2.1. Software Encoder Energy Analysis

Figure 4.15 shows our energy analysis during the real-time encoding of an HD video. We find that a significant portion of the system energy (39.6%) is spent on ME. ME is the most memory-intensive unit in the encoder [345], accounting for 43.1% of the total encoding cycles. The next major contributor to total system energy is the deblocking filter, which we discuss in in Section 4.5.2. While intra-prediction, transform, and quantization are other major contributors to the energy consumption, none of them individually account for more than 9% of the total energy. The remaining energy, labeled *Other* in the figure, is spent on decoding the encoded frame, and behaves the same way as the software decoding we discuss in Section 4.5.2.

We find that 59.1% of encoder energy goes to data movement. The majority of this data movement is generated by ME, and accounts for 21.3% of the total system energy. Other major contributors to data movement are the MC and deblocking filter used to decode the encoded frames (see Section 4.5.2).

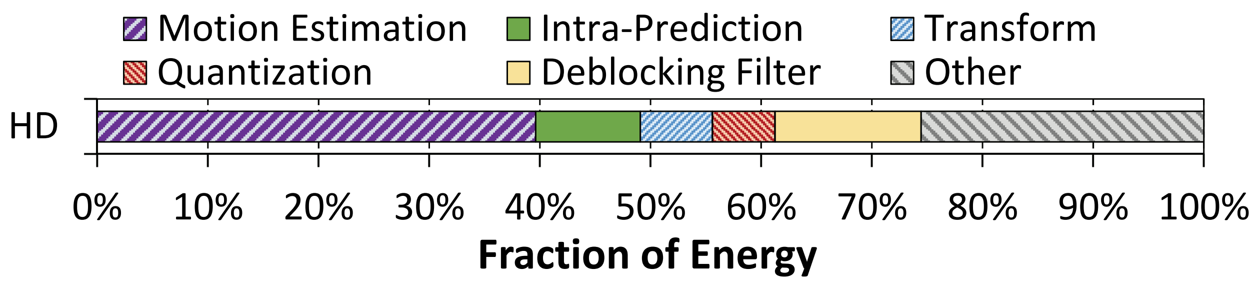


Figure 4.15. Energy analysis of VP9 software encoder.

4.6.2.2. Analysis of PIM Effectiveness

In libvpx, ME uses the diamond search algorithm [396] to locate matching objects in reference frames, using the sum of absolute differences (SAD) to evaluate matches. ME is highly memory-intensive and requires high memory bandwidth. 54.7% of ME's energy consumption is spent on data movement (not shown). For each macro-block, ME needs to check for matching objects in *three* reference frames, which leads to a large amount of data movement. Hence, ME is a good candidate for PIM execution.

While ME is more compute- and memory-intensive than the other PIM targets we examine, it primarily calculates SAD, which requires only simple arithmetic operations. As a result, ME can be executed with high performance on our PIM core (see Section 4.2.3), and can be implemented as a fixed-function PIM accelerator that is similar in design to the ME component of the hardware VP9 decoder. Our analysis shows that our PIM accelerator occupies an area of 1.24 mm^2, which requires no more than 35.4% of the area available per vault for PIM logic (see Section 4.2.3). Thus, we conclude that it is feasible to implement ME using PIM in a consumer device with 3D-stacked memory.

4.6.3. VP9 Hardware Encoder

In this section, we present our analysis on the VP9 hardware encoder. While software encoding offers more flexibility, hardware encoding enables much faster encoding in an energy-efficient manner. Similar to the hardware video decoder (Section 4.5.3), the hardware video encoder employs prefetching to hide memory latency. Unlike the decoder, the encoder's memory access pattern is highly predictable, as the search window for each reference frame is predefined. As a result, the encoder (specifically the ME unit) successfully hides latency. However, energy consumption and the cost of data movement remain as major challenges.

4.6.3.1. Hardware Encoder Energy Analysis

Figure 4.16 shows the breakdown of off-chip traffic for encoding a single frame of 4K and HD video, both with and without lossless frame compression. Similar to our findings for the software encoder, the majority of the off-chip traffic is due to the reference frame pixels fetched by ME (shown as *Reference Frame* in the figure), which accounts for 65.1% of the entire encoder's data movement for HD video. The traffic grows significantly when we move to 4K video, as the encoder requires 4.3x the number of pixels read during HD video encoding. While frame compression can reduce the amount of data transferred by 59.7%, a significant portion of encoder energy is still spent on reference frame traffic.

The two other major contributors to off-chip traffic are (1) the current frame after encoding, and (2) the reconstructed frame, which is used as a future reference frame. The current frame generates 14.2% of the traffic when frame compression is disabled, but since compression cannot be applied to the encoded version of the current frame, the total data movement for the current frame takes up to 31.9% of the traffic when compression is enabled. The reconstructed frame consumes on average 12.4% of the traffic across the two resolutions.

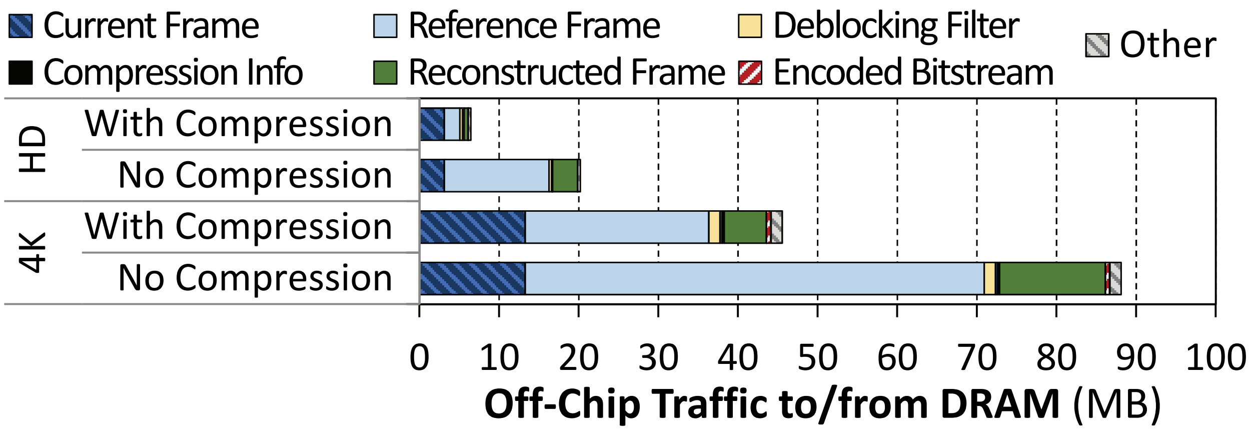


Figure 4.16. Off-chip traffic breakdown of VP9 hardware encoder.

4.6.3.2. Analysis of PIM Effectiveness

Similar to the software encoder, ME is responsible for the majority of data movement in the hardware encoder, and is again a good candidate for PIM execution, which would eliminate the need to move three reference frames to the on-chip encoder. Figure 4.17 shows the high-level architecture of a modified VP9 encoder with in-memory ME (which is the same as the PIM accelerator in Section 4.6.2.2). Once in-memory ME (❶ in the figure) generates the motion vector for a macro-block, it sends the vector back to the mode decision unit (❷) in the on-chip accelerator. Note that we also offload MC (❸) and the deblocking filter (❹) to PIM, similar to the hardware decoder (see Section 4.5.3.2), since they use reference frames (❺) together with the motion vectors output by ME to reconstruct the encoded frame. This eliminates data movement during frame reconstruction. We conclude that data movement can be reduced significantly by implementing the ME, MC, and deblocking filter components of the VP9 hardware encoder in memory.

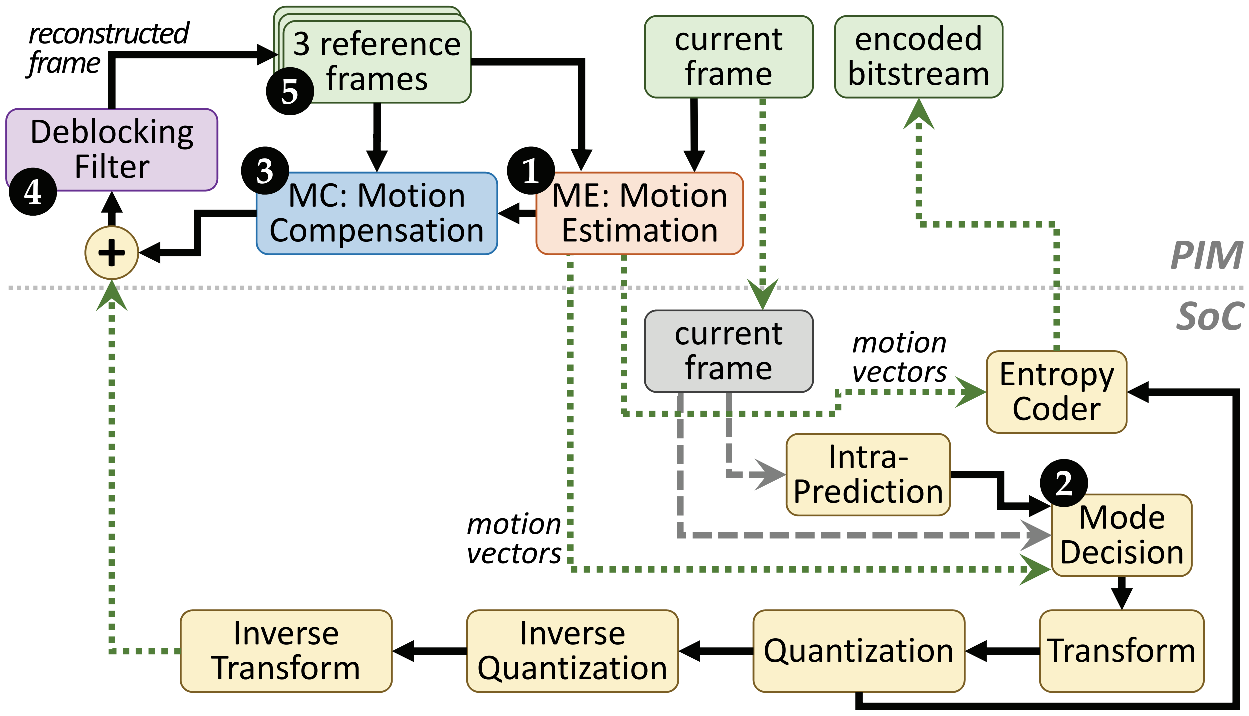


Figure 4.17. Modified VP9 encoder with in-memory ME.

4.7. System Integration

We briefly discuss two system challenges to implementing PIM in the consumer device context.

4.7.1. Software Interface for Efficient Offloading

We use a simple interface to offload the PIM targets. We identify PIM targets using two macros, which mark the beginning and end of the code segment that should be offloaded to PIM. The compiler converts these macros into instructions that we add to the ISA, which trigger and end PIM kernel execution. Examples are provides in a related work [104].

4.7.2. Coherence Between CPUs and PIM Logic

A major system challenge to implementing PIM is CPU-PIM communication and coherence. The majority of the PIM targets we identified are minimal functions/primitives that are interleaved with other parts of the workloads in a fine-grained manner. As a result, to fully benefit from offloading them to PIM, we need efficient communication between the CPU and PIM logic to allow coordination, ordering, and synchronization between different parts of the workloads. We find that coherence is required to (1) enable efficient synchronization, (2) retain programmability, and (3) retain the benefits' of using PIM. As a result, we employ a simple fine-grained coherence technique, which uses a local PIM-side directory in the logic layer to maintain coherence between PIM cores (or PIM accelerators), and to enable low-overhead fine-grained coherence between PIM logic and the CPUs. The CPU-side directory acts as the main coherence point for the system, interfacing with both the processor caches and the PIM-side directory. Our design can benefit further from other approaches to coherence like LazyPIM [44, 104].

4.8. Methodology

We evaluate our proposed PIM architectures using the gem5 [38] full-system simulator. Table 7.1 lists our system configuration.

SoC	4 OoO cores, 8-wide issue; *L1 I/D Caches*: 64 kB private, 4-way assoc.; *L2 Cache:* 2 MB shared, 8-way assoc.; *Coherence*: MESI
PIM Core	1 core per vault, 1-wide issue, 4-wide SIMD unit, *L1 I/D Caches*: 32 kB private, 4-way assoc.
3D-Stacked Memory	2 GB cube, 16 vaults per cube; *Internal Bandwidth:* 256 GB/s; *Off-Chip Channel Bandwidth:* 32 GB/s
Baseline Memory	LPDDR3, 2 GB, FR-FCFS scheduler

Table 4.1. Evaluated system configuration.

We organize our PIM architecture such that each vault in 3D-stacked memory has its own PIM logic. The PIM logic for each vault consists of either a PIM core or a PIM accelerator. We assume that the PIM core is ISA-compatible with the main processor cores in the SoC, and is a simple general-purpose core (e.g., 1-wide issue, no prefetcher) that has only a 32KB private L1 cache. Each PIM accelerator has a small 32kB buffer to hold its working set of data. Like prior works [14, 15, 44, 97, 167, 168], we use the memory bandwidth available to the logic layer of 3D-stacked memory as the memory bandwidth to the PIM core/accelerator. We model our 3D-stacked memory similar to HBM [186] and use the bandwidth available in the logic layer of HBM (256GB/s), which is 8x more than the bandwidth available (32GB/s) to the off-chip CPU cores.

To evaluate how PIM benefits each of the PIM targets that we identified, we study each target in isolation, by emulating each target separately, constructing a microbenchmark for the component, and then analyzing the benefit of PIM in our simulator. We use Verilog RTL [258] to estimate the area overhead of the VP9 hardware encoder and decoder.

Chrome Browser. For texture tiling, we precisely emulate `glTextImage2d()` for OpenGL from the Intel i965 graphics driver, using 512x512-pixel RGBA tiles as the input set. For color blitting, we construct a microbenchmark that closely follows the color blitting implementation in Skia, and use randomly-generated bitmaps (ranging from 32x32 to 1024x1024 pixels) as inputs. For compression and decompression, we use LZO [283] from the latest upstream Linux version. We generate the input data by starting Chrome on our Chromebook, opening 50 tabs, navigating through them, and then dumping the entire contents of the Chromebook's main memory to a file.

TensorFlow Mobile. For packing, we modify the *gemmlowp* library [116] to perform only matrix packing, by disabling the quantized matrix multiplication and result matrix unpacking routines. We evaluate in-memory quantization using a microbenchmark that invokes TensorFlow Mobile's post-Conv2D/MatMul re-quantization routines, and we use the result matrix sizes of GEMMs to reflect real-world usage.

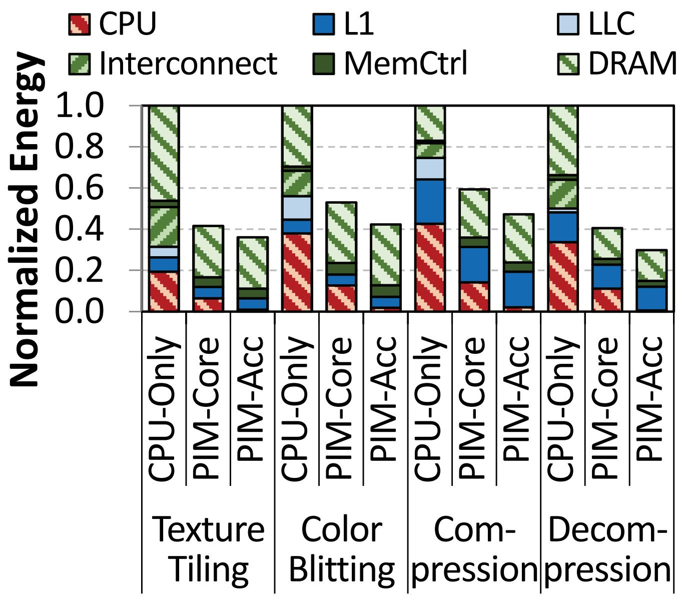
Video Playback and Capture. For sub-pixel interpolation and the deblocking filter, we construct microbenchmarks that closely follow their implementations in *libvpx*, and use 100 frames from a 4K Netflix video [382] as an input. For motion estimation, we develop a microbenchmark that uses the diamond search algorithm [396] to perform block matching using three reference frames, and use 10 frames from an HD video [382] as an input. For the hardware VP9 evaluation, we use the off-chip traffic analysis (Figures 4.12 and 4.16) together with in-memory ME and in-memory MC traffic analysis from our simulator to model the energy cost of data movement.

4.9. Evaluation

In this section, we examine how our four consumer workloads benefit from PIM. We show results normalized to a processor-only baseline (*CPU-Only*), and compare to PIM execution using PIM cores (*PIM-Core*) or fixed-function PIM accelerators (*PIM-Acc*).

4.9.1. Chrome Browser

Figure 4.18 shows the energy consumption and runtime of CPU-Only, PIM-Core, and PIM-Acc across different browser kernels, normalized to CPU-only. We make three key observations on energy from the left graph in Figure 4.18. First, we find that offloading these kernels to PIM reduces the energy consumption, on average across all of our evaluated web pages, by 51.3% when we use PIM-Core, and by 61.0% when we use PIM-Acc. Second, we find that the majority of this reduction comes from eliminating data movement. For example, eliminating data movement in texture tiling contributes to 77.7% of the total energy reduction. Third, PIM-Acc provides 18.9% more energy reduction over PIM-Core, though the energy reduction of PIM-Acc is limited by the highly-memory-intensive behavior of these browser kernels. This is because while computation on PIM-Acc is more efficient than computation using PIM-Core, data movement accounts for the majority of energy consumption in the kernels, limiting the impact of more efficient computation on overall energy consumption.



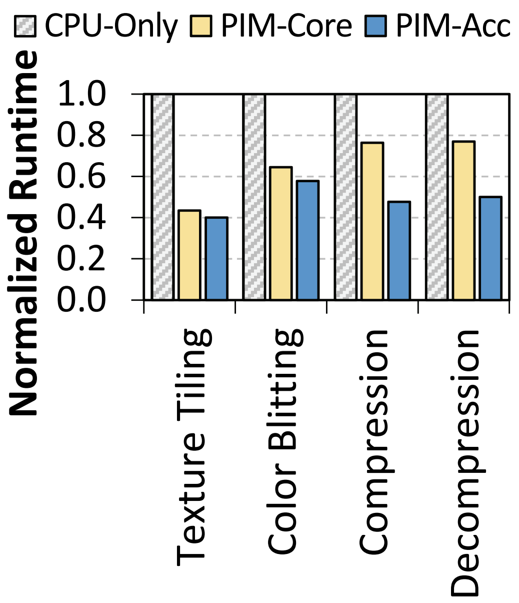


Figure 4.18. Energy (left) and runtime (right) for all browser kernels, normalized to CPU-Only, for kernel inputs listed in Section 4.8.

We make five observations on performance from the right graph in Figure 4.18. First, PIM-Core and PIM-Acc significantly outperform CPU-Only across all kernels, improving performance by 1.6x and 2.0x, on average across all of our evaluated web pages. Second, we observe that the browser kernels benefit from the higher bandwidth and lower access latency of 3D-stacked memory (due to their data-intensive nature), leading to performance improvement. Third, this performance benefit increases as the working set size increases. For example, our analysis (not shown) shows that the speedup increases by 31.2% when the texture size increases from 256x256 to 512x512 pixels. Fourth, the performance improvement of PIM-Acc over PIM-Core for texture tiling and color blitting is limited by the low computational complexity of these two kernels. Fifth, compression and decompression benefit significantly from using PIM-Acc over PIM-Core in terms of performance, because these kernels are more compute-intensive and less data-intensive compared to texture tiling and color blitting.

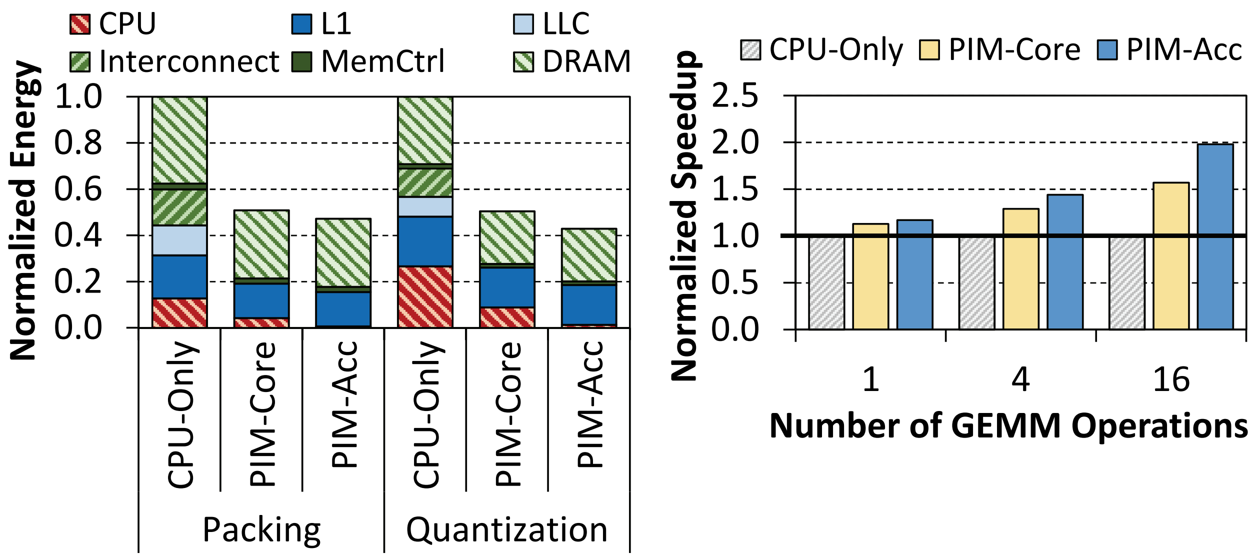


Figure 4.19. Energy (left) and performance (right) for TensorFlow Mobile kernels, for four neural networks [157, 339, 354, 360].

4.9.2. TensorFlow Mobile

Figure 4.19 (left) shows the energy consumption of CPU-Only, PIM-Core, and PIM-Acc for the four most time- and energy-consuming GEMM operations for each input neural network in packing and quantization, normalized to CPU-Only. We make three key observations. First, PIM-Core and PIM-Acc decrease the total energy consumption by 50.9% and 54.9%, on average across all four input networks, compared to CPU-Only. Second, the majority of the energy savings comes from the reduction in data movement, as the computation energy accounts for a negligible portion of the total energy consumption. For instance, 82.6% of the energy reduction for packing is due to the reduced data movement. Third, we find that the data-intensive nature of these kernels and their low computational complexity limit the energy benefits PIM-Acc provides over PIM-Core.

Figure 4.19 (right) shows the total execution time of CPU-Only, PIM-Core and PIM-Acc as we vary the number of GEMM operations performed. For CPU-Only, we evaluate a scenario where the CPU performs packing, GEMM operations, quantization, and unpacking. To evaluate PIM-Core and PIM-Acc, we assume that packing and quantization are handled by the PIM logic, and the CPU performs GEMM operations. We find that as the number of GEMM operations increases, PIM-Core and PIM-Acc provide greater performance improvements over CPU-Only.

For example, for one GEMM operation, PIM-Core and PIM-Acc achieve speedups of 13.1% and 17.2%, respectively. For 16 GEMM operations, the speedups of PIM-Core and PIM-Acc increase to 57.2% and 98.1%, respectively, over CPU-Only. These improvements are the result of PIM logic (1) exploiting the higher bandwidth and lower latency of 3D-stacked memory, and (2) enabling the CPU to perform GEMM in parallel while the PIM logic handles packing and quantization. For example, offloading packing to the PIM core speeds up the operation by 32.1%. Similar to our observation for the browser kernels (Section 4.9.1), as these kernels are not compute-intensive, the performance improvement of PIM-Acc over PIM-Core is limited.

4.9.3. Video Playback and Capture

4.9.3.1. VP9 Software

Figure 4.20 (left) shows the energy consumption of CPU-Only, PIM-Core, and PIM-Acc across different video kernels, normalized to CPU-Only. We make three observations from the figure. First, all of the kernels significantly benefit from PIM-Core and PIM-Acc, with an average energy reduction of 46.8% and 66.6%, respectively, across our input videos. Second, the energy reductions are a result of the memory-bound behavior of these video kernels. Moving the kernels to PIM logic (1) eliminates a majority of the data movement, and (2) allows the kernels to benefit from the lower power consumed by PIM logic than by the CPU without losing performance. For example, offloading sub-pixel interpolation and the deblocking filter to PIM-Core reduces their total energy consumption by 49.9% and 50.1%, respectively. Of those reductions, 34.6% and 51.9% come from the reduced energy consumption of PIM-Core compared to the SoC CPU, while the rest of the reduction comes from decreasing data movement. Third, we find that PIM-Acc provides, on average, a 42.7% additional energy reduction than PIM-Core across all of the video kernels.

Figure 4.20 (right) shows the runtime of CPU-Only, PIM-Core, and PIM-Acc on our video kernels. We observe from the figure that PIM-Core and PIM-Acc improve performance over CPU-Only by 23.6% and 70.2%, respectively, averaged across all input videos. While PIM-Core provides a modest speed up (12.6%) over CPU-Only for the motion estimation kernel, PIM-Acc significantly outperforms CPU-Only and improves performance by 2.1x. The reason is that motion estimation

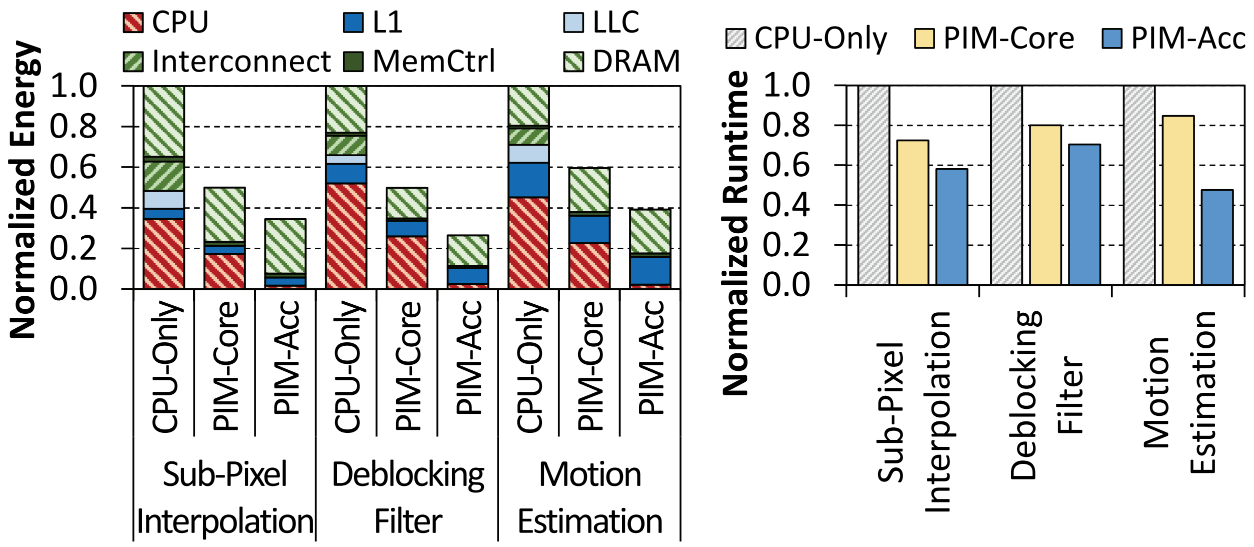


Figure 4.20. Energy (left) and runtime (right) for all video kernels, normalized to CPU-Only, using 100 4K frames [382] for decoding and 10 HD [382] frames for encoding.

kernel has relatively high computational demand, compared to the other PIM targets we evaluate, and thus, benefits much more from executing on PIM logic.

4.9.3.2. VP9 Hardware

Figure 4.21 shows the total energy consumption for the decoder (left) and encoder (right), for three configurations: VP9 only (which uses the baseline VP9 hardware accelerator [372] in the SoC), VP9 with PIM-Core, and VP9 with PIM-Acc. For each configuration, we show results without and with lossless frame compression.

We make four key observations from the figure. First, we find that for both the VP9 decoder and encoder, off-chip data movement is the major source of energy consumption, consuming 69.2% and 71.5% of the total decoder and encoder energy, respectively. Second, we find that computation using the VP9 hardware is an order of magnitude more energy-efficient than computation using PIM-Core. As a result, when compression is enabled, PIM-Core actually consumes 63.4% *more* energy than the VP9 baseline. Third, we find that PIM-Acc reduces energy when compared to the VP9 hardware baseline, by 75.1% for decoding and by 69.8% for encoding. This is because PIM-Acc embeds parts of the VP9 hardware itself in memory, retaining the computational energy efficiency of the baseline accelerator while reducing the amount of data movement that takes place.

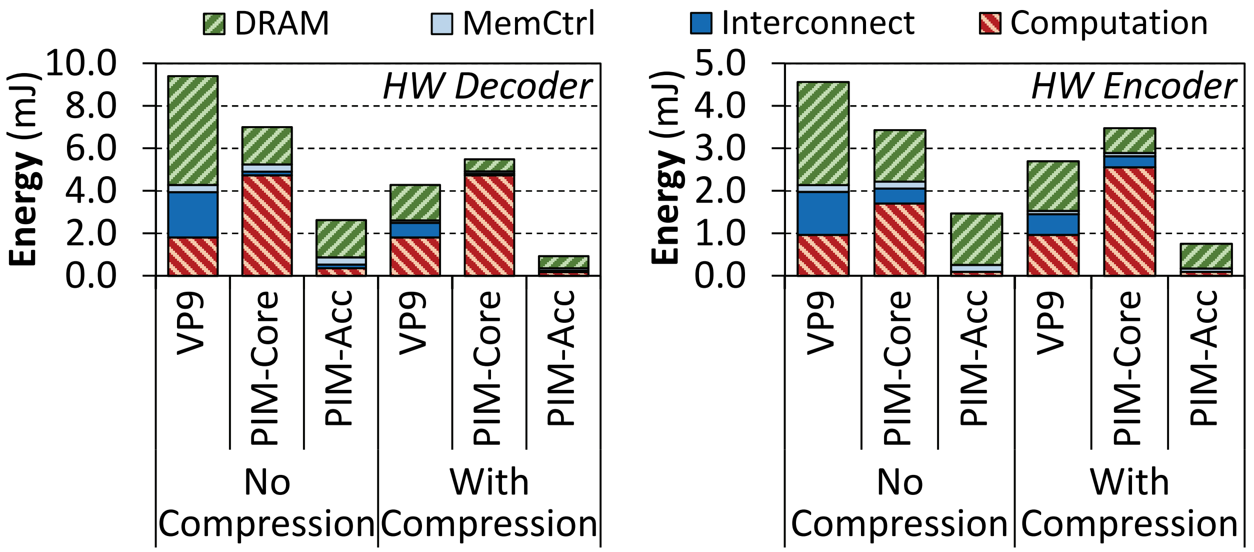


Figure 4.21. Total energy for VP9 hardware decoder (left) and VP9 hardware encoder (right), based on an analysis using 4K and HD video [382].

Fourth, we find that PIM-Acc without compression uses less energy than the VP9 hardware baseline with compression. This indicates that the PIM logic is more effective at reducing data movement than compression alone. We achieve the greatest energy reduction by combining PIM-Acc with compression.

4.10. Summary

Energy is a first-class design constraint in consumer devices. In this chapter, we analyze several widely-used Google consumer workloads, and find that data movement contributes to a significant portion (62.7%) of their total energy consumption. Our analysis reveals that the majority of this data movement comes from a number of simple functions and primitives that are good candidates to be executed on low-power processing-in-memory (PIM) logic. We comprehensively study the energy and performance benefit of PIM to address the data movement cost on Google consumer workloads. Our evaluation shows that offloading simple functions from these consumer workloads to PIM logic, consisting of either simple cores or specialized accelerators, reduces system energy consumption by 55.4% and execution time by 54.2%, on average across all of our workloads. We conclude that reducing data movement via processing-in-memory is a promising approach to improve both the performance and energy efficiency of modern consumer devices with tight area, power, and energy budgets.

Chapter 5

CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators

Modern systems increasingly employ specialized accelerators. Accelerators offer a way to improve system performance and energy efficiency in the face of diminishing returns from process technology scaling [86, 327]. Many recent works (e.g., [91, 108, 132, 148, 153, 155, 196, 217, 246, 252, 253, 378, 398]) propose *on-chip* customized hardware accelerators. These accelerators aim to satisfy the demands of emerging workloads.

Recent advances in 3D-stacked memory technology enable the practical implementation of *near-data processing*, where computational logic is placed close to memory. In particular, *near-data accelerators* (NDAs) can further boost the performance and energy benefits that conventional accelerators promise, by reducing the amount of *data movement* between the processor and the main memory [13,14,15,42,77,93,97,99,139,167,169,207,248,275,319,320,321,342,356,381,389,392].

Despite the significant benefits of accelerators, system challenges remain a main stumbling block to the mainstream adoption of specialized accelerators. The lack of an efficient communication mechanism between CPUs and accelerators creates a significant overhead to synchronize data updates between the two [147, 285, 298, 329, 330, 350, 366]. This inefficiency generates unnecessary data movement, which can negate the benefits of accelerators. In addition, programmers are often required to use custom-designed communication mechanisms between CPU cores and accelerators, which leads to high programming complexity. These challenges can be largely mitigated by making the accelerator *coherent* with the rest of the system, as by doing so: (1) developers can use the conventional shared memory programming model, allowing them to use well-known synchronization mechanisms to coordinate between the accelerators and the CPUs; and (2) accelerators can efficiently share data with each other and with the rest of the system, instead of relying on bulk data transfers [222, 329, 330].

In contrast to coherence for *on-chip* accelerators [222, 298, 350], coherence for NDAs, which reside off-chip, is significantly more challenging for two reasons. First, the energy and performance costs of *off-chip* communication between NDAs and CPUs are very high. For example, for the Hybrid Memory Cube [174], the off-chip serial links consume as much energy to move data as the DRAM array consumes to access the data [16, 184, 295]. In fact, the energy and performance costs of *off-chip* communication between NDAs and CPUs are orders of magnitude greater than the costs of on-chip communication [203]. Second, target applications for NDAs are fundamentally different from those for on-chip accelerators, as NDA applications typically have low computational demand, suffer from poor locality, and generate a large amount of off-chip data movement [14, 15, 16, 42, 89, 163, 169, 275, 392]. These applications incur a large number of coherence misses. Given these challenges, it is impractical for an NDA to utilize traditional coherence mechanisms (e.g., MESI [110, 291]), which would need to send off-chip messages for *every cache miss* to coherence management hardware (e.g., a coherence directory) that reside on-chip with the CPU. The high cost and frequency of these messages with a traditional mechanism would eliminate most, if not all, of the benefits of near-data acceleration.

Our goal is to address the coherence challenge for near-data accelerators (NDAs). Enabling coherence for NDAs provides two key benefits: (1) programmers can use the well-known traditional shared memory model to program systems with NDAs, and (2) we can simplify how NDAs communicate and share data with the rest of the system.

5.1. Summary of Key Insights and Observations

While some recent NDA proposals acknowledge the need for NDA–CPU coherence [15, 93, 97, 380], these works largely sidestep the issue by assuming that the NDA and CPU share only a limited amount of data. While this is true for some NDA applications [93, 99, 206, 302, 380], our application analysis indicates that *this is not the case* for many other important NDA applications. We comprehensively analyze two important classes of such applications: graph processing frameworks and hybrid in-memory databases. We make a *key observation* that not all portions of these applications benefit from being executed on the NDA, and the portions that remain on the CPU (called *CPU threads*) often *concurrently* access the same region of data (e.g., graphs, database data structures) as the portions executed on the NDA (called *NDA kernels*), leading to *significant data sharing*. To understand the characteristics of sharing, we delve further into the memory access patterns of the CPU threads and NDA kernels, and we make a *second key observation*: while CPU threads and NDA kernels share the same data regions, they typically *do not* collide concurrently on (i.e., simultaneously access) the same cache lines. Furthermore, in the rare case of collisions, the CPU threads only read those cache lines, meaning that they *rarely* update the same data that an NDA is actively working on.

Using the major insights we obtain from our application analysis, we perform a design space exploration and examine three existing approaches for coherence: non-cacheable regions [14, 77, 93, 99, 275, 302], coarse-grained coherence [93, 97, 235, 361, 380, 389], and fine-grained coherence [42]. We find that (1) all three approaches eliminate a significant portion of the potential benefits of NDAs, as they generate a large amount of off-chip coherence traffic, and in some cases prevent concurrent execution of CPUs and NDAs; and (2) the majority of off-chip data movement (i.e., coherence traffic) generated by these approaches is unnecessary, primarily because the approaches pessimistically assume that *every* memory access needs to acquire coherence permissions or that shared data *cannot* be cached. We find that much of this unnecessary off-chip coherence traffic can be eliminated if the coherence mechanism has insight into what part of the shared data is *actually* accessed by NDA kernels and CPU threads. Unfortunately, this insight is *not* available before execution for many workloads with irregular access patterns. For example, in many pointer chasing or graph processing workloads, the path taken during pointer or graph traversal is *not* known prior to execution [14, 169, 216].

Based on these observations, we find that an *optimistic* approach to coherence can address the challenges of NDA coherence. An optimistic execution model for NDA enables us to gain insight into the memory accesses before any coherence permissions are checked, and thus, enforce coherence with only the *necessary* data movement. To this end, we propose *Coherence for Near-Data Accelerators* (CoNDA), a mechanism that lets the NDA *optimistically* start execution assuming that it has coherence permissions, without issuing *any* coherence messages off-chip. CoNDA executes NDA kernels in portions to keep hardware overheads low (see Section 5.4.5). While optimistically executing a portion of an NDA kernel, CoNDA records the memory accesses inside the NDA to gain insight into what part of the shared data is *actually* accessed by the kernel. During optimistic execution, CoNDA does not commit any data updates. When CoNDA finishes optimistic execution for the NDA kernel portion, it exploits the recorded information to check which coherence operations are necessary, in order to avoid off-chip data movement for unnecessary coherence operations. If the NDA kernel portion does not need coherence operations for *any* of its data updates, CoNDA commits the data updates. If the NDA kernel portion actually requires any coherence operations, CoNDA invalidates the uncommitted data updates, performs the needed coherence operations, and re-executes the NDA kernel portion.

Our optimistic execution model is inspired by Optimistic Concurrency Control (OCC) [223], which was first proposed in the database community and later harnessed for various purposes (e.g., Transactional Memory [23, 150, 161, 162, 263, 332], enforcing sequential consistency [51], deterministic shared memory [72]). The optimistic execution model fits very well within the context of NDA coherence for three reasons. First, the optimistic approach makes it possible to identify the *necessary* coherence traffic, by gaining insight into the memory accesses performed by the NDA and by the rest of the system before generating off-chip coherence requests. Second, our application analysis shows that CPU threads *rarely* update the same data that an NDA is actively working on (which is also true for other applications that do not have a high degree of data sharing [93, 99, 206, 302, 380]). This behavior leads to a very low re-execution rate of NDA kernels, which is one of our key motivations behind adopting an optimistic execution model for NDA coherence. Third, NDAs are often relatively simple fixed-function accelerators or small programmable cores that lack sophisticated ILP techniques [14, 15, 42, 77, 97, 169, 248, 275, 302, 361, 380]. As a result, the cost of kernel re-execution on an NDA is much lower than that on a sophisticated out-of-order CPU core.

We find that CoNDA is highly effective in efficiently enforcing coherence between the CPU threads and NDAs. Compared to three major existing coherence mechanisms, CoNDA improves performance by 19.6% over the highest performance prior mechanism and reduces memory system energy by 18.0% over the most energy-efficient prior mechanism, on average across our 16-thread workloads operating on modest data set sizes. Over a CPU-only system with no NDAs, CoNDA improves performance by 66.0% and reduces memory system energy consumption by 43.7%. Over an NDA-only system, CoNDA improves performance by 51.8%. These benefits arise because CoNDA eliminates the majority of unnecessary coherence traffic, and is able to retain almost all of the benefits of near-data acceleration. CoNDA comes within 10.4% and 4.4% of the performance and energy, respectively, of an ideal NDA mechanism that incurs no penalty for coherence.

5.2. Motivation

Enabling coherence for near-data accelerators provides two key benefits: (1) programmers can use the well-known traditional shared memory model to program systems with near-data accelerators, and (2) we can simplify how accelerators communicate and share data with each other and with the rest of the system. In this section, we analyze the data sharing characteristics of several important data-intensive workloads (Section 5.2.2), and then study how existing coherence mechanisms perform for these workloads (Section 5.2.3).

5.2.1. Baseline Architecture

Figure 5.1 shows the baseline organization of the architecture we assume in this work, which includes programmable or fixed-function near-data accelerators (NDAs). Each NDA executes an *NDA kernel* that is invoked by the CPU threads. In our evaluation, we implement programmable

NDAs consisting of *in-order* cores that are ISA-compatible with the CPU cores, but that do not have large caches or any sophisticated ILP techniques. Each NDA includes small private L1 I/D caches. While our evaluations use a programmable NDA, CoNDA can be used with *any* programmable, fixed-function, or reconfigurable NDA.

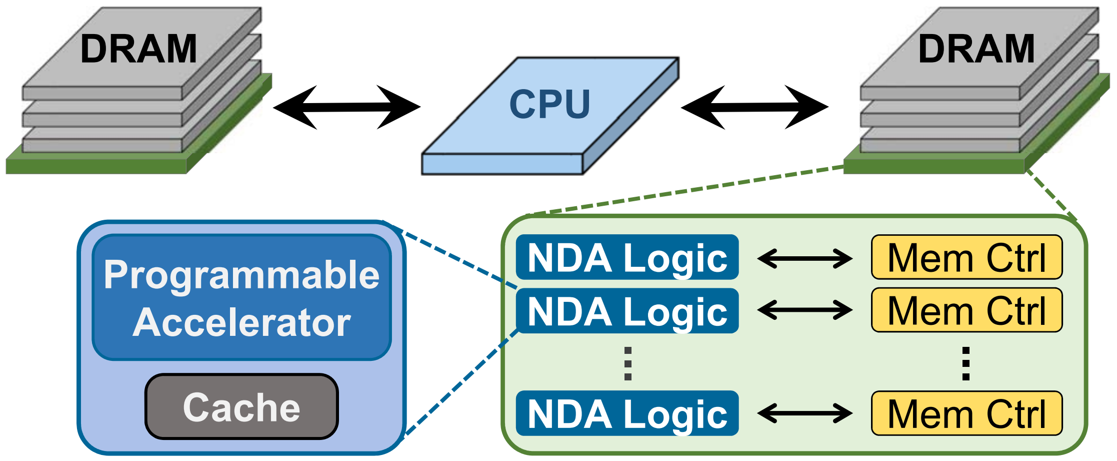


Figure 5.1. High-level organization of our NDA architecture.

5.2.2. Application Analysis

An application can benefit from near-data acceleration when its memory-intensive parts are offloaded to NDAs. The memory-intensive parts of an application generate a significant amount of data movement, and often exhibit poor temporal locality, leading to high execution times and energy consumption on a CPU. In the NDA, such application parts can benefit from the high-bandwidth, low latency and low-energy memory access available in 3D-stacked memory. However, the compute-intensive parts of the application should *remain on the CPU cores* to maximize performance [15, 42, 97, 167, 275, 320], especially if they can exploit the CPU cache hierarchy well or benefit from sophisticated ILP techniques.

Sharing Data Between NDAs and the CPU. Because many applications have compute-intensive parts that should be executed on the CPU, NDA kernels and CPU threads may share data with each other, depending on how an application is partitioned. While some applications [93,99,206,302,380] can be partitioned to limit this data sharing, we make a *key observation* that *this is not the case* for many important classes of applications.

A major example is multithreaded graph processing frameworks, such as Ligra [336], where multiple threads operate in parallel on the same shared in-memory graph [215, 336, 384]. Each thread executes a graph algorithm, such as *PageRank* [46]. We rigorously study a number of these algorithms [336] and find that when we carefully convert each of them for NDA execution, only some portions of each algorithm are well-suited for NDA, while the remaining portions perform better if they stay on the CPU. With this partitioning, the CPU portion of each thread executes on the CPU cores while the NDA portions (sometimes concurrently) execute on the near-data accelerators, with all of the threads sharing the graph and other intermediate data structures. For example, we find that for the *Radii* and *PageRank* algorithms running on the *arXiv* input graph, 60.1% and 71.0% of last-level cache (LLC) misses generated by the CPU threads access the shared data (e.g., the graph, intermediate data structures) after we convert these algorithms for NDA execution.

A second major example is modern in-memory databases. Today, analytical and transactional operations are combined into a single hybrid transactional/analytical processing (HTAP) database system [25, 30, 205, 226, 255, 257, 317]. The analytical queries of these hybrid databases are well-suited for NDA execution, as they have long execution times (i.e., the queries run for long enough to amortize the overhead of dispatching the query to the NDA) and touch a large number of rows, leading to a large amount of random memory accesses and data movement [217, 261, 380]. In contrast, even though transactional queries access the same data, they perform better if they stay on the CPU, as they have short execution times, are latency-sensitive, and have cache-friendly access patterns. In such workloads, concurrent accesses from both NDA kernels and CPU threads to shared data structures are inevitable.

Many NDA workloads exhibit the same characteristics that we find in these applications. In several workload domains, recent works show that only parts of an application benefit from NDA execution, while the rest of the application should stay on CPUs [15, 42, 97, 167, 275, 320]. These NDA kernels often share many data structures with the CPU threads. For example, in TensorFlow Mobile [128], prior work shows that two functions (*packing* and *quantization*) should be NDA kernels [42]. These kernels share the neural network data structures (e.g., matrices) with the CPU threads, which execute functions such as convolution and matrix multiplication.

Shared Data Access Patterns. To further understand data sharing, we analyze the memory access patterns of the CPU threads and NDA kernels. We make a *second key observation*: while CPU threads and NDA kernels share data structures, they often *do not concurrently access the same elements* (e.g., a shared graph node or database row) in these structures. For example, when we analyze the *Connected Components* and *Radii* algorithms from Ligra [336] using the *arXiv* input graph, only 5.1% and 7.6%, respectively, of the CPU accesses collide with (i.e., access the same cache line as) accesses from the NDA. Across all of our applications, only 11.2% of these collisions (less than 1% of all accesses) are writes from a CPU thread, and all other collisions are CPU thread reads.

The low rate of collisions with data updates (i.e., writes) is intuitive from a programming perspective, because when programmers offload some of the code to an accelerator, they try to avoid having the CPU thread update the data that the accelerator is working on. Such behavior is a common characteristic of many accelerator-centric applications [91, 182, 196, 217, 252, 306].

5.2.3. Analysis of NDA Coherence Mechanisms

We explore three types of existing mechanisms that can be used to enforce coherence between CPU threads and NDAs.

Non-Cacheable Approach. One approach to sidestep coherence is to mark any data accessed by the NDA (i.e., the NDA data region) as *non-cacheable* by the CPU [14]. This ensures that any CPU writes are immediately visible to the NDA. While this works well for applications where the CPU *rarely* accesses the NDA data region, it performs poorly for many applications where the CPU accesses the region *often*. For Ligra [336] applications with a representative input graph (*arXiv*), we find that 38.6% of all accesses made by CPU threads to memory are to the NDA data region.

Figures 5.2 and 5.3 show the memory system energy consumption and speedup of different coherence mechanisms for a system with NDAs, normalized to a *CPU-only* baseline where the entire application runs on the CPU. In the figures, we also show a mechanism called *Ideal-NDA*, where there is no energy or performance penalty for coherence. We observe from the figures that the non-cacheable approach (*NC*) fails to provide any energy savings (and, in fact, greatly increases energy consumption), and on average performs 6.0% worse than CPU-only. Therefore, NC is a poor fit for applications where the NDA and CPU threads share data, as it is unable to realize any benefit from NDA in our workloads.

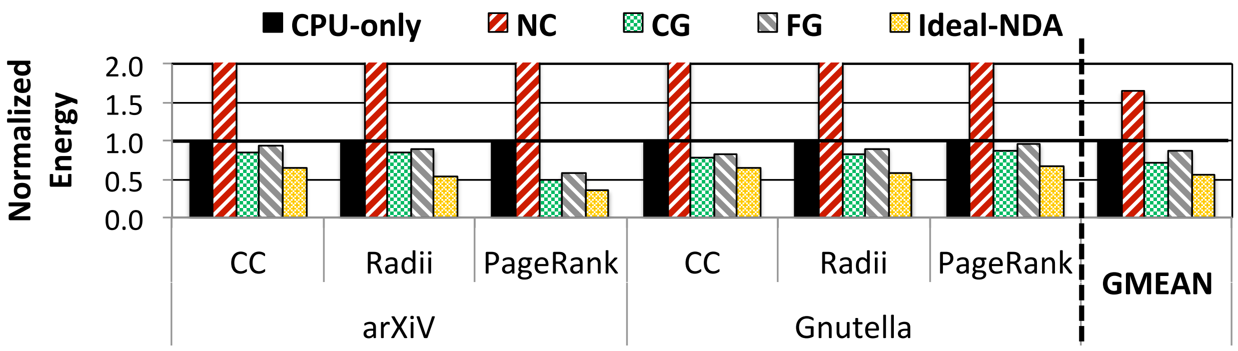


Figure 5.2. Memory system energy with existing coherence mechanisms.

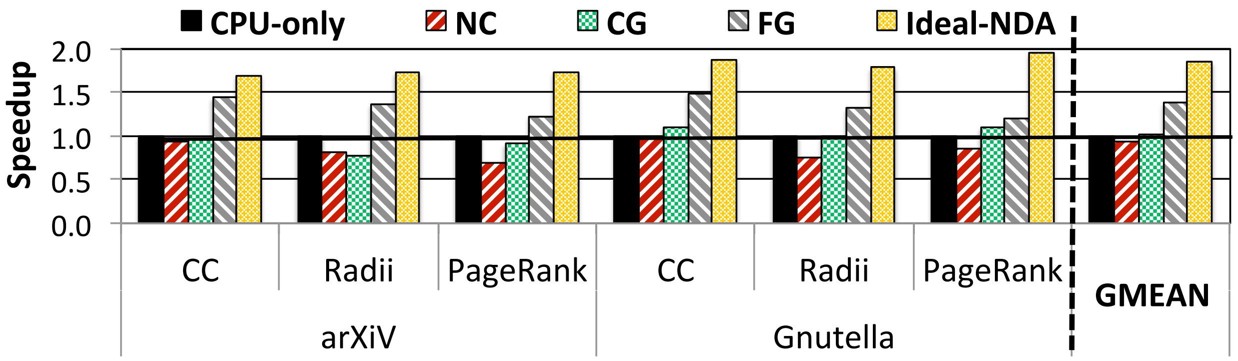


Figure 5.3. Speedup with existing coherence mechanisms.

Coarse-Grained Coherence. A second approach is *coarse-grained coherence*, where there is a *single* coherence permission that applies to the *entire* NDA data region. This works well when there is only a limited amount of data shared between the CPU threads and the NDA kernel, but incurs high overheads when there is a high amount of data sharing and there is no easy way to find what part of the data will be accessed by the NDA before execution (e.g., due to irregular access patterns). The CPU must flush *all* dirty cache lines in the NDA data region *every time* an NDA acquires coherence permissions for the region, *even if the NDA does not access most of the data in the region*. This results in a significant amount of *unnecessary* data movement. For example, with only four CPU threads, *PageRank* flushes 227x the number of cache lines actually accessed by the NDA. While coherence at a smaller granularity (e.g., one entry per page [97]) can reduce this overhead in some cases, the overhead remains high for the many applications that have irregular access patterns. For example, pointer chasing applications [14, 169, 216] access only a few cache lines in each page, but still require *all* changes to the page to be flushed.

Some instances of coarse-grained coherence use *coarse-grained locks* to provide the CPU or NDA with *exclusive* access to a coarse-grained region. Without exclusive access, data can ping-pong between the CPU and the NDA when they concurrently access the NDA data region, increasing data movement. Coarse-grained locks avoid ping-ponging by having the NDA acquire exclusive access to a region for the duration of the NDA kernel. Our analysis shows that coarse-grained locks greatly limit performance when *any* sharing exists, by forcing CPU threads and NDA kernels to serialize. Averaged across all Ligra applications running on a representative input graph (*Gnutella*), coarse-grained locks block 87.9% of the CPU's memory accesses during NDA execution. As Figures 5.2 and 5.3 show, the high impact of unnecessary flushes and serialization cause coarse-grained locks (*CG* in the figures) to eliminate a large portion of the energy and performance benefits that Ideal-NDA provides. In fact, *CG* performs 0.4% *worse* than CPU-only, on average.

We conclude that while CG works well in some cases, it is *not* suitable for many important NDA applications.

Fine-Grained Coherence. Traditional, or *fine-grained*, coherence protocols (e.g., MESI [110, 291]) have two major qualities well suited for applications with irregular memory accesses (e.g., graph workloads, databases, pointer chasing). First, fine-grained coherence acquires coherence permissions for only the pointers that are actually traversed during pointer chasing. The path taken during pointer traversal is not known ahead of time. As a result, even though a thread often accesses only a few dispersed pieces of the data structure, a coarse-grained mechanism has no choice but to acquire coherence permissions for the entire data structure. Fine-grained coherence allows the CPU or NDA to acquire permissions for only the pieces of data in the data structure that are actually accessed. Second, fine-grained coherence can ease programmer effort when developing applications for NDAs, as multithreaded programs already use this programming model.

Unfortunately, if an NDA participates in fine-grained coherence with the CPU, it has to exchange coherence messages for *every cache miss* with the CPU directory over a narrow pin-limited bus.

Since NDA kernels are memory-intensive and exhibit poor temporal locality, this fine-grained message exchange generates large amounts of off-chip data movement. A large amount of those coherence messages are unnecessary, as our analysis from Section 5.2.2 shows that the vast majority of NDA accesses do not collide with CPU writes and, thus, do not need coherence.

We apply a number of optimizations to fine-grained coherence to reduce the amount of off-chip data movement. We enforce exclusive ownership of each cache line by either the NDA or the CPU. We add a local NDA directory in DRAM to maintain coherence between multiple NDAs. We add a bit to each entry in both the CPU and NDA directories, to mark the cache lines that are owned by an NDA. If a coherence miss occurs for a cache line owned by the NDA, this means that the CPU also does not have a copy of the cache line, and no request needs to be sent to the CPU. Unfortunately, even with these optimizations, fine-grained coherence using the MESI protocol [110, 291] (*FG*) eliminates a significant portion of the energy and performance benefits of Ideal-NDA, as shown in Figures 5.2 and 5.3. This is because the optimized FG (1) still needs to send off-chip coherence requests to acquire ownership; (2) is unable to avoid the majority of off-chip coherence requests, as the NDA kernel has a high number of cache misses that generate many coherence requests; and (3) still causes data to ping-pong between the CPU and the NDA, generating many off-chip coherence messages for a single cache line.

We conclude that while FG acquires permissions for only cache lines that are actually accessed, it still causes a lot of unnecessary off-chip data movement.

Limitations of Existing Coherence Mechanisms. The majority of unnecessary off-chip data movement generated by existing coherence mechanisms is due to a lack of *insight* into when coherence requests are actually necessary during NDA kernel execution. Without having any such insight, existing mechanisms preemptively issue coherence requests, many of which are *not needed*, causing significant off-chip data movement to maintain coherence. As a result, none of these mechanisms can exploit the potential energy and performance benefits of NDAs.

5.3. Optimistic NDA Execution

As we see in Section 5.2, existing coherence mechanisms can eliminate the benefits of near-data acceleration for many important application classes. We find that the majority of off-chip data movement generated by these coherence mechanisms is unnecessary, primarily because the mechanisms pessimistically assume that (1) *every* memory operation needs to acquire coherence permissions or (2) data shared between CPUs and NDAs *cannot* be cached. Much of this unnecessary movement can be eliminated if a coherence mechanism has insight on *what part of the shared data* is *actually* accessed. Based on our observations from Section 5.2, we propose to use *optimistic execution* for NDAs. When executing in optimistic mode, an NDA gains insight into its memory accesses by tracking the accesses *without* issuing any coherence requests. When optimistic execution is done, the NDA uses the tracking information to perform *necessary* coherence requests for *only* the parts of the shared data that were *actually* accessed during execution, which minimizes coherence-related data movement.

In this section, we discuss our optimistic execution model and how it retains existing memory consistency guarantees (Section 5.3.1), and analyze when coherence messages are necessary in optimistic mode (Section 5.3.2). We assume a sequentially-consistent memory model in this section, but our execution model can easily be applied to other common memory consistency models, such as the x86-TSO model (Section 5.4.8).

In Section 5.4, we propose CoNDA, a coherence mechanism for NDAs that makes use of optimistic execution.

5.3.1. Execution Model

An application running on the CPU can issue a call to start executing code on an NDA. When the NDA starts executing, the CPU threads may execute concurrently. The NDA executes in optimistic mode, where it assumes that it *always* has coherence permissions on the cache lines that it uses, *without* checking the CPU coherence directory. This avoids the need for off-chip coherence communication during execution. Because the NDA has not actually checked coherence, it ensures that none of its data updates are committed to memory during optimistic execution.

When optimistic execution stops, the system must determine whether it can *commit* the data updates from optimistic execution. This requires the NDA to determine if any coherence requests should have been issued to guarantee correctness. To keep the commit mechanism simple, our execution model makes use of *coarse-grained atomicity*, where *all* memory updates by the NDA are treated as if they all occur *at the moment when optimistic execution stops*. Thus, for all memory operations that take place while the CPU threads and the NDA execute concurrently, the CPU thread memory operations are effectively ordered *before* the NDA memory operations, which is a valid ordering for sequential consistency. When the NDA attempts to commit, the system first checks to see if any coherence violation happened (see Section 5.3.2). If no violation occurred, the NDA data updates are committed. Otherwise, the NDA must resolve any violations by performing the necessary coherence operations, and re-execute the optimistically-executed code. Because CPU threads and NDA kernels rarely access the same cache lines during concurrent execution (see Section 5.2.2), re-execution happens rarely, as we show in Section 5.6.2, making optimistic execution efficient.

We do *not* expose the optimistic execution behavior of the NDA to programmers. Our execution model ensures that memory consistency is not violated, and thus, the programmer can treat the system like a conventional multithreaded system. To make use of near-data acceleration, a programmer simply needs to insert macros to demarcate portions of the application that should be executed on the NDA (see Section 5.4.1), and can treat the NDA kernel as just another thread. As is the case for concurrent CPU threads, the programmer assumes that (1) instructions across multiple threads can be interleaved *in any order* acceptable under the memory consistency model of the system, and (2) they need to use synchronization primitives if they want to enforce a *specific* ordering. Our mechanism supports synchronization primitives (see Section 5.4.7).

5.3.2. Identifying Necessary Coherence Requests

In order to maintain coherence, the NDA must perform any *necessary* coherence operations before committing its uncommitted memory operations. Coherence requests are necessary *only*

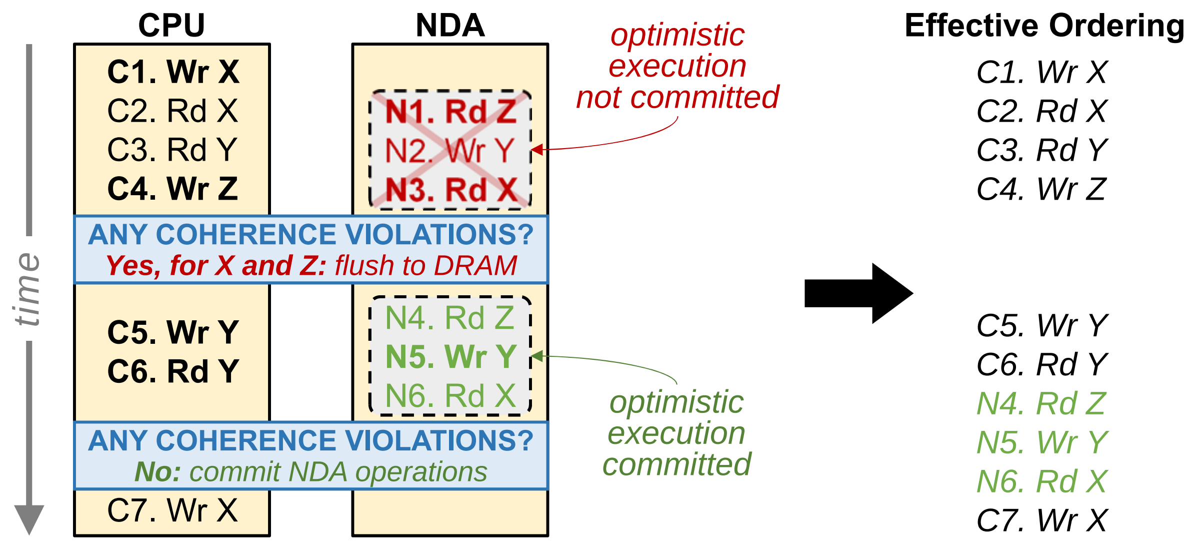


Figure 5.4. Example timeline of optimistic NDA execution.

when the NDA and/or a CPU thread update a cache line that both the NDA and the CPU thread access. Figure 5.4 shows an example of how CPU and NDA operations are ordered. We discuss three possible interleavings of CPU and NDA memory operations to the same cache line, and show that only one of the interleavings leads to a violation and requires NDA re-execution to ensure correct execution.

Case 1: NDA Read and CPU Write (to the Same Cache Line). *This case triggers re-execution.* As we discuss in Section 5.3.1, all NDA operations are ordered to take place *once optimistic execution stops*. As a result, any CPU memory accesses that take place *during* optimistic mode are ordered before any NDA operations. For example, the NDA read (e.g., N1 in Figure 5.4, which reads cache line Z) should be ordered after the CPU write to the same cache line (e.g., C4). However, since the NDA did not issue a coherence request before performing the read operation, it did not read the updated value that was written by the CPU. In order to maintain coherence, the value written by the CPU must be flushed to DRAM, and the NDA must re-execute to ensure correct ordering (e.g., N1–N3 are re-executed as N4–N6).

Note that the CPU may perform the write (e.g., C1 to cache line X) *before* optimistic execution begins. This can still require re-execution if the updated cache line is not written back to memory before the NDA performs the read (e.g., N3).

Case 2: NDA Write and CPU Read (to the Same Cache Line). *This case does not trigger re-execution.* In our execution model, any read to a cache line by the CPU during optimistic mode execution is ordered *before* the NDA's write to the same cache line. As a result, the CPU should *not* read the value written by the NDA. For example, NDA write N5 to cache line Y in Figure 5.4 is effectively ordered to take place after CPU read C6 to the same cache line. To ensure that the CPU does not read the value written by the NDA, the NDA maintains the updated value in a *data update buffer* until it can confirm that there is no need for it to re-execute instructions. Once this is confirmed, the NDA commits the write to memory, and invalidates any stale copies of the data in the CPU cache.

If the programmer wants to *guarantee* that the CPU read sees the NDA's write, the program must use synchronization primitives to enforce this ordering (see Section 5.4.7). This is true in conventional CPU multithreading as well.

Case 3: NDA Write and CPU Write (to the Same Cache Line). *This case does not trigger re-execution.* Similar to Case 1, the NDA write (e.g., N5) takes place after the CPU write (e.g., C5), and the NDA holds the update in the data update buffer. However, when the NDA is ready to commit its updates (i.e., there is no need to re-execute), it cannot simply flush the old cache line in the CPU. This is because the CPU and the NDA may have written to *different words* in the same cache line. To ensure that no updates are lost, the data update buffer must include a per-word dirty bit mask, as in [235]. When the system commits the NDA write, it first retrieves the latest version of the cache line from the CPU, and then overwrites only the words in the cache line that were written to by the NDA using the values of those words that are in the data update buffer. Again, if the programmer wishes to enforce a specific ordering of the writes, they must use a synchronization primitive (e.g., a write fence), as in conventional CPU multithreading.

5.4. CoNDA Architecture

In this section, we describe *Coherence for Near-Data Accelerators* (CoNDA), an efficient coherence mechanism that makes use of optimistic NDA execution (see Section 5.3) to avoid unnecessary off-chip coherence traffic. Figure 5.5 shows the high-level operation of CoNDA. In

CoNDA, when an application wants to launch an NDA kernel (Section 5.4.1), the NDA begins executing the kernel in optimistic mode (❶ in Figure 5.5; Section 5.4.2). While the NDA kernel executes, all CPU threads continue to execute normally, and *never* make use of optimistic execution.

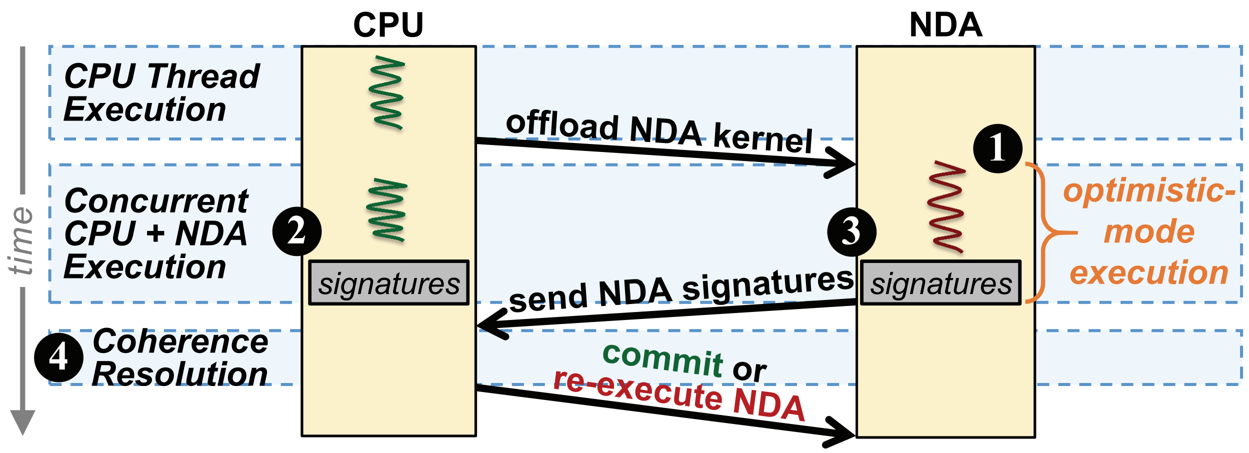


Figure 5.5. High-level operation of CoNDA.

To gain the insight needed to perform only the necessary coherence requests, CoNDA efficiently tracks the addresses of all NDA reads, NDA writes, and CPU writes during optimistic execution using *signatures* (❷ and ❸; Section 5.4.3). Once optimistic execution starts, any NDA data updates are initially flagged as uncommitted (Section 5.4.4). These updates cannot be committed until all necessary coherence requests are performed. When optimistic execution is done (Section 5.4.5), CoNDA attempts to resolve coherence (❹; Section 5.4.6). The NDA transmits its signatures to the CPU, and CoNDA compares the NDA signatures with the CPU signatures to identify the necessary coherence requests. If CoNDA detects any coherence violation (see Section 5.3.2), (1) the NDA invalidates all of its uncommitted updates; (2) the CPU resolves the coherence requests, performing only the necessary coherence operations (including any cache line updates); and (3) the NDA re-executes the uncommitted portion of the kernel. Otherwise, performs the necessary coherence operations, clears the uncommitted flag for all data updates in the NDA L1 cache (i.e., any uncommitted data updates are committed), and resumes optimistic execution if the NDA kernel is not finished.

To preserve the conventional multithreaded programming interface, CoNDA correctly supports synchronization primitives (Section 5.4.7). CoNDA enables optimistic execution, ensures correctness, and reduces unnecessary coherence traffic compared to state-of-the-art coherence mechanisms, all with a low hardware overhead (Section 5.4.9).

5.4.1. Program Interface

We provide a simple interface for programmers to port applications to CoNDA. The programmer identifies the portion(s) of the code to execute on an NDA using two macros (`NDA_begin` and `NDA_end`). The compiler converts the macros into instructions that we add to the ISA, which *trigger* and *end* NDA kernel execution, respectively. To reduce tracking overhead, CoNDA needs to know which pages in memory *might* be accessed by an NDA, which we call the *NDA data region*. The NDA data region can be identified by a compiler, or can be manually annotated by the programmer. CoNDA stores this information by adding one bit to each page table entry, which when set indicates that CoNDA needs to track reads from and writes to this page during optimistic execution.

5.4.2. Starting Optimistic NDA Execution

CoNDA starts optimistic execution when an NDA kernel is launched, or after a successful coherence resolution operation. When an NDA kernel is first launched, the CPU dispatches the kernel's starting PC and any live-in registers to a free NDA. Any time optimistic execution starts, CoNDA takes a checkpoint of the current NDA state, which consists of the NDA's PC and software-visible registers. This checkpoint is used in case commit fails and the NDA needs to re-execute.

5.4.3. Signatures

CoNDA uses signatures to track whether any coherence requests are necessary during optimistic execution. CoNDA uses this information once optimistic execution ends to perform *only* the necessary coherence requests.

Implementation. To reduce the amount of storage needed to track coherence requests without missing any addresses that need to be checked for correct coherence, we implement signatures in CoNDA using fixed-length parallel Bloom filters [40]. In a parallel Bloom filter, an N-bit signature is partitioned into M segments. Each segment in the signature employs a unique hash function

(H_3 [316]). When an address is added to the signature, each segment's hash function maps the address to a single bit in the segment, which we call the *hashed value*, and the bit is set to 1. Using a parallel Bloom filter, CoNDA can perform three operations. First, it can check if a filter contains a given memory address, by generating the hashed values for the address and checking if they are set in *all M* segments. Second, it can retrieve a list of all addresses in the filter, by using the signature expansion technique [51, 52]. Third, it can quickly compare two filters to see if both filters might contain one or more of the same addresses, by taking the bitwise AND of the two signatures to generate the filter intersection and checking that each segment in the intersection has at least one bit set (indicating at least one matching address that is shared by both filters).

Once a bit is set in a parallel Bloom filter segment, the bit remains set until the filter is reset. As a result, there are *no false negatives*. This ensures that CoNDA checks all of the addresses that are added to the signatures during optimistic execution. Due to aliasing of some hashed values, Bloom filters can introduce a limited number of false positives (see below), which may lead to unnecessary re-executions without affecting correctness. The parallel Bloom filters in CoNDA are reset every time optimistic execution starts.

Tracking Memory Operations During NDA Execution. CoNDA maintains three sets of signatures during optimistic execution, as shown in Figure 5.6. The `NDAReadSet` records the addresses of all cache lines read from by the NDA. The `NDAWriteSet` records the addresses of all cache lines written to by the NDA. The `CPUWriteSet` records the addresses of all cache lines *in the NDA data region* that (1) a CPU thread writes to during optimistic execution, or (2) have dirty copies in a CPU cache before an NDA kernel starts. The CPU scans the caches before launching an NDA kernel to find dirty cache lines that reside in the NDA data region. Our evaluation shows that across the entire program, the total time spent on scanning accounts for less than 1% of the overall execution time (Section 5.6), including TLB overheads. This is because the NDA kernels are very data intensive, and require orders of magnitude more time to execute than scanning the CPU L1 cache tag stores, which can be done in parallel across caches.

In the unlikely case that scanning becomes a performance bottleneck (which we never observe in our experiments), we can optimize the scan operation by introducing a Dirty-Block Index [318] to track dirty NDA data.

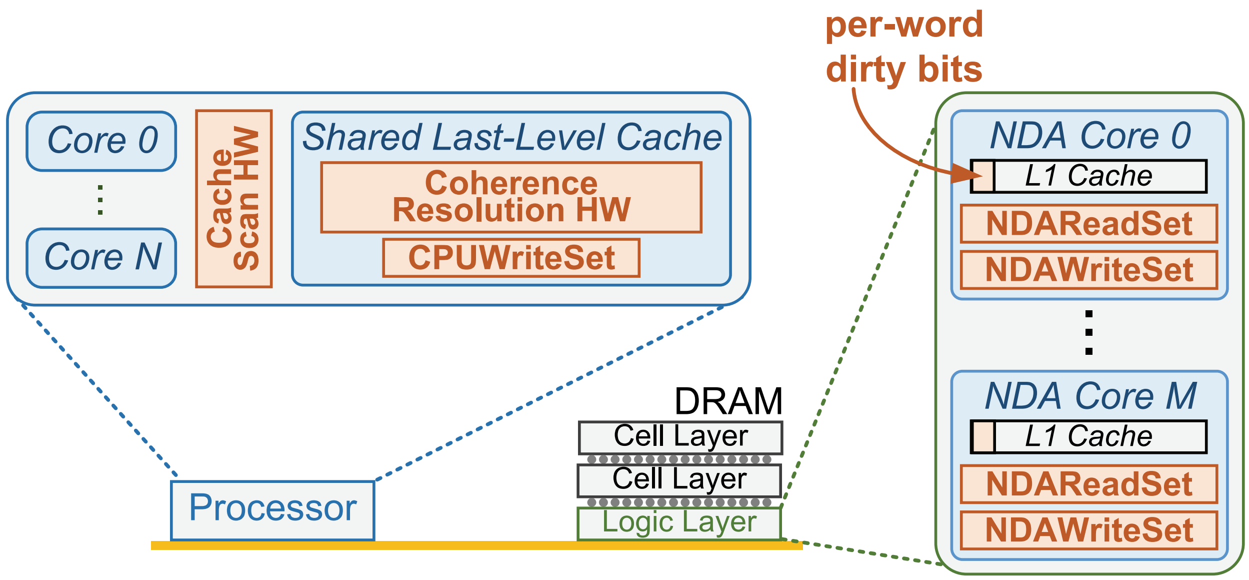


Figure 5.6. Hardware additions to support CoNDA.

Signature Size. Each Bloom filter is composed of a fixed-length register. The size of the register determines the maximum number of addresses can be added to the filter without exceeding a given false positive rate. Once the maximum number of addresses is reached in any signature, CoNDA stops optimistic execution and begins resolving necessary coherence requests for the NDA kernel. We size the signatures in CoNDA to balance the storage overhead, the false positive rate (as false positives can cause unnecessary re-executions), how frequently optimistic execution must be stopped, and the amount of data movement needed to transfer signatures from the NDA to the CPU when CoNDA performs any necessary coherence requests.

The `NDAReadSet` and `NDAWriteSet` each use one 256 B register that is split into four segments. The `CPUWriteSet` uses *eight* 256 B Bloom filters. This is because the `CPUWriteSet` does not need to be transmitted off-chip during coherence resolution, so we can use multiple filters to increase the addresses that the `CPUWriteSet` can hold. We use a round robin policy to select which of the filters an address is added to. We target a 20% false positive rate (worst case) in CoNDA, which allows us to hold up to 250 addresses in each 256 B filter. Our Bloom filter analysis is based on mathematical derivations in prior work [316].

5.4.4. Buffering Uncommitted Values

Each NDA includes a small private L1 data cache. During optimistic execution, when an NDA performs a write operation, the write value cannot be committed to memory until CoNDA can perform the necessary coherence operations. As a result, the NDA buffers the updated data value in the L1 cache, using techniques similar to prior works on speculative execution [23, 51, 149, 150, 161, 162, 219, 223, 263, 346]. We add a per-word dirty bit mask [235] to each cache line, as shown in Figure 5.6, to mark all uncommitted data updates.

5.4.5. Ending Optimistic NDA Execution

CoNDA dynamically determines when to end optimistic execution. Execution in optimistic mode stops when one of three events occurs: (1) the NDA reaches the end of the NDA kernel, (2) the NDA L1 data cache needs to evict a speculative cache line, or (3) one of the signatures cannot hold more addresses without exceeding the false positive rate. To determine if a signature cannot hold more addresses, CoNDA maintains a counter for the number of addresses added to each signature since optimistic execution started.

5.4.6. Identifying Necessary Coherence Requests

Once optimistic execution ends, CoNDA uses the signatures to check for and resolve any necessary coherence operations, and attempts to commit any uncommitted NDA data updates. The `NDAReadSet` and `NDAWriteSet` are sent from the NDA to *coherence resolution* logic residing in the CPU. The coherence resolution logic then calculates the intersection of the `NDAReadSet` and the `CPUWriteSet`. There are two cases, depending on whether the intersection contains a match.

Case 1 (Conflict). If the intersection contains a match, a coherence violation *may* have occurred (i.e., an NDA read and a CPU write may have been to the same address; see Section 5.3.2). This means that the pending NDA data updates cannot be committed, as they *may* violate correct memory ordering. Instead, CoNDA must perform the necessary coherence operations and then re-execute the NDA operations. To do so, the CPU flushes any dirty cache lines that match addresses in the `NDAReadSet` to DRAM, and places a copy of these cache lines in the NDA L1 cache. The CPU uses the signature expansion method [51,52] to decode the `CPUWriteSet` into corresponding addresses, and then checks if those addresses are present within the `NDAReadSet`. Once the dirty cache line flush completes, the coherence resolution logic sends a message to the NDA that the commit attempt failed. The NDA invalidates all uncommitted data in its L1 cache, rolls back to its checkpointed state, and restarts optimistic execution. If the same portion of the NDA kernel fails commit N times (we empirically set N to three), CoNDA guarantees forward progress of the NDA by acquiring a lock for each cache line in the `NDAReadSet`(e.g., by temporarily setting the page table read-only bit for any page that contains a cache line in the `NDAReadSet`). This ensures that the kernel does not roll back anymore, and avoids livelock.

Case 2 (No Conflict). If no match is found in the intersection of the `NDAReadSet` and the `CPUWriteSet`, then no memory ordering violation exists, and the commit process starts to perform any necessary coherence operations. First, the coherence resolution hardware computes the intersection of the `NDAWriteSet` and the `CPUWriteSet`. If an overlap is found, CoNDA uses signature expansion (see Section 5.4.3) to identify which cache lines need to be merged (see Section 5.3.2), and sends these cache lines to the NDA for merging, which ensures write-after-write coherence. Second, all cache lines in the CPU cache that match an address in the `NDAWriteSet` are invalidated. Third, a message is sent to the NDA, which clears the uncommitted flag on all NDA cache lines and allows the lines to be written to DRAM. Finally, if the NDA kernel is not finished, the NDA continues executing the kernel by starting optimistic execution from the instruction after the commit.

During coherence resolution, CPU threads continue to execute, but all coherence directory entries for cache lines in the NDA data region are locked to ensure atomicity. If a thread accesses any cache line in the NDA region, the thread stalls until the coherence resolution completes.

5.4.7. Support for Synchronization Primitives

CoNDA allows a programmer to use traditional synchronization primitives to manually order memory operations and provide atomicity. When an NDA reaches a synchronization primitive (e.g., `Acquire`, `Release`), it performs three steps. First, CoNDA ends optimistic execution, and commits any pending speculative updates before the primitive. This ensures that there are no remaining memory operations that *must* be ordered before the synchronization primitive, preserving the memory ordering expected by the programmer. Second, the NDA executes the primitive *non-speculatively*, performing any coherence operations necessary for the primitive. By executing the primitive non-speculatively, CoNDA guarantees that the primitive *cannot be rolled back*, allowing the primitive to work exactly as it does in conventional CPU multithreading. Third, after the primitive has been performed, the NDA resumes optimistic execution.

5.4.8. Support for Weaker Consistency Models

In Section 5.3.1, we assume a sequentially-consistent memory model, but CoNDA can support other common memory consistency models. We illustrate this using a brief case study on how CoNDA works with the x86-TSO (total store ordering) consistency model. CoNDA can support x86-TSO by waiting to add store addresses to signatures until the addresses are issued to the memory system. To support x86-TSO, each CPU and NDA needs to include a FIFO write buffer that can hold issued stores until they are written to the memory system. Since the write buffer is architecturally invisible, CoNDA ensures that the `CPUWriteSet` signatures do not record an address that is in the write buffer. CoNDA records a write address in the `CPUWriteSet` only when the address leaves the write buffer and is sent to the memory system. The coherence resolution logic in CoNDA remains unchanged, as the write must become visible to the entire system when it is completed by the memory system. For example, if the CPU writes to memory address A, and an NDA reads from memory address A, there is no conflict as long as the write stays in the write buffer, and the NDA read can complete, which is an expected memory ordering in x86-TSO. Once the write from the write buffer is issued to the memory, the address is recorded in the `CPUWriteSet` and becomes visible to the entire system.

5.4.9. Hardware Overhead

Each NDA uses 512 B to store signatures, while the CPU uses 2 kB in total. Aside from the signatures, CoNDA's overhead consists mainly of (1) 1 bit per page in the page table (0.003% of DRAM capacity) and 1 bit per TLB entry for the page table flag bits (Section 5.4.1); (2) a 1.6%

increase in NDA L1 data cache size for the per-word uncommitted data bit mask (Section 5.4.4); and (3) two 8-bit counters per NDA to track the number of addresses stored in each signature.

5.5. Methodology

We implement CoNDA in the gem5 simulator [38]. We perform all simulations in full-system mode using the x86 ISA, and modify the integrated DRAMSim2 [75] DRAM timing simulator to model 3D-stacked HMC DRAM [105, 174] available to the NDAs. To accurately model the coherence mechanisms, we modify the Ruby memory model in gem5. We include a local coherence directory for the NDAs, and set the CPU coherence directory as the main point of coherence for the system. Both the NDA and CPU directories use the MESI protocol. Table 7.1 shows our system configuration.

Processor	16 cores, 8-wide issue, 2 GHz frequency, out-of-order *L1 I/D Caches*: 64 kB private, 4-way, 64 B blocks *L2 Cache* 4 MB shared, 8-way, 64 B blocks *Coherence*: MESI
Near-Data Accelerator	16 NDAs per stack, 1-wide issue, 2 GHz frequency, in-order *L1 I/D Caches*: 64 kB private, 4-way, 64 B blocks *Coherence*: MESI
HMC [174]	one 4 GB cube, 16 vaults per cube, 16 banks per vault
Memory	DDR3-1600, 4 GB, FR-FCFS scheduler

Table 5.1. Evaluated system configuration for CoNDA.

Our simulation platform models all of the overheads of CoNDA. During coherence resolution, these overheads include (1) 20 cycles to send each signature from the NDA to the CPU, which is a conservative estimate given that it takes 3 cycles to transfer a 256 B signature across the HMC link; (2) 2 cycles to compare a CPU signature to an NDA signature, which is more conservative than prior work [51, 52]; (3) 8 cycles to invalidate a CPU cache line on a matching address; and (4) 12 cycles to transfer a cache line from the CPU to the NDA to merge writes. We model the full overhead of NDA kernel re-execution, which involves (1) invalidating uncommitted cache lines and erasing signatures; (2) rolling back the NDA to a checkpoint, (3) resolving coherence, and (4) re-running the kernel. We assume that NDA rollback takes 8 cycles, since the NDA is a small core with a small cache, making the rollback significantly cheaper than for large out-of-order CPUs.

We report *memory system energy* using an energy model similar to prior work [42], which accounts for the total energy consumed by the DRAM, on-chip and off-chip interconnects, and all caches. We use detailed simulator statistics to drive this model. We model the 3D-stacked DRAM energy as the energy consumed per bit, leveraging estimates and models from prior work [184]. We estimate the energy consumption of all L1 and L2 caches using CACTI-P 6.5 [270], assuming a 22 nm process. We model the off-chip interconnect using the method used by prior work [97], which estimates the HMC SerDes energy consumption as 3 pJ/bit for data packets.

Applications. We study two classes of applications that are well-suited for NDA. For these applications, we use two types of input datasets: (1) a modest size dataset, which we use to perform the majority of our evaluations; and (2) a large size dataset, which we use in Section 5.6.5 to show how CoNDA's benefits scale as the input sizes increase.

We evaluate three graph applications from Ligra [336] (a lightweight multithreaded graph framework): *Connected Components* (*CC*), *Radii*, and *PageRank* (*PR*). The modest size dataset for our graph applications consists of input graphs from three real-world networks [344]: *Enron* email communication network (73384 nodes, 367662 edges), *arXiv* General Relativity (10484 nodes, 28984 edges), and peer-to-peer *Gnutella25* (45374 nodes, 109410 edges). The large size dataset consists of input graphs from two real-world networks [344], which are on average 14.8x larger than the graphs used in the modest size dataset: (1) the *Amazon* product network (334863 nodes, 925872 edges), and (2) the *DBLP* collaboration network (317080 nodes, 1049866 edges)).

We also evaluate an in-house prototype of an in-memory database (IMDB) that supports HTAP workloads [257,317,349]. Our IMDB uses a state-of-the-art, highly-optimized hash join kernel [358]. For the modest size dataset, we simulate an IMDB system with 64 tables, 64K tuples per table, and 32 randomly-populated integer fields per table. Our transactional workload consists of 64K transactions, where each transaction reads from or writes to 1–3 randomly-chosen database tuples. Our two analytical workloads consist of 128 or 256 analytical queries (*HTAP-128* and *HTAP-256*) that use the *select* and *join* operations. For the large size dataset, we increase the size of the IMDB system to 64 tables with 640K tuples per table, with 256K transactions in the transactional workload and 1024 queries in the analytical workload (*HTAP-1024*).

Identifying NDA Kernels in Applications. Similar to prior works [15, 42], we identify candidate NDA kernels that are memory-intensive and not cache friendly. Using hardware performance counter data, we consider a function to be an NDA kernel candidate when (1) it is memory intensive (i.e., its LLC misses per kilo instruction, or MPKI, is greater than 20 [212, 213, 269]); (2) the majority of the function's execution time is spent on data movement; and (3) it is one of the top three functions in the workload in terms of execution time. From this set of candidate kernels, we select kernels for NDA execution such that we minimize the amount of data sharing that occurs between CPU threads and NDA kernels. A lower amount of data sharing leads to higher performance under existing coherence mechanisms (i.e., NC, CG, FG), and more modest performance improvements with CoNDA. We manually annotate the NDA data region, by replacing all `malloc` calls to data in the region with a custom memory allocator called `nda_alloc`, which notifies gem5 that the data belongs to the region.

In most Ligra applications, we select the *edgeMap* function as an NDA kernel. This function processes and updates a subset of edges for each vertex [336], which generates many random memory accesses. In our *HTAP* workloads, we select the analytical queries (i.e., *select* and *join* operations) as NDA kernels.

5.6. Evaluation

We show results normalized to a *CPU-only* baseline, and compare CoNDA to NDA execution using fine-grained coherence (*FG*), coarse-grained locks (*CG*), non-cacheable NDA data (*NC*), or ideal coherence (*Ideal-NDA*), as described in Section 5.2.3.

5.6.1. Off-Chip Data Movement

Figure 5.7 shows the normalized off-chip data movement (which we measure as bytes transferred between the NDAs and the CPU) of the NDA coherence mechanisms for a system with 16 CPU cores and 16 NDAs. We make three observations from the figure. First, CoNDA significantly reduces the *overall* data movement compared to all prior NDA coherence mechanisms, with an average reduction of 30.9% over the next best mechanism, CG. Compared to CG, which has to flush every dirty cache line in each region acquired by the NDA, CoNDA uses its insight on NDA memory accesses to greatly reduce the number of lines flushed (e.g., by 92.2% for Radii using arXiv). Second, NC's data movement is very high because *all* processor accesses to the NDA data region must go to DRAM. Third, CoNDA reduces data movement by 86.3% over CPU-only, because it successfully allows memory-intensive portions of the applications to no longer consume off-chip bandwidth.

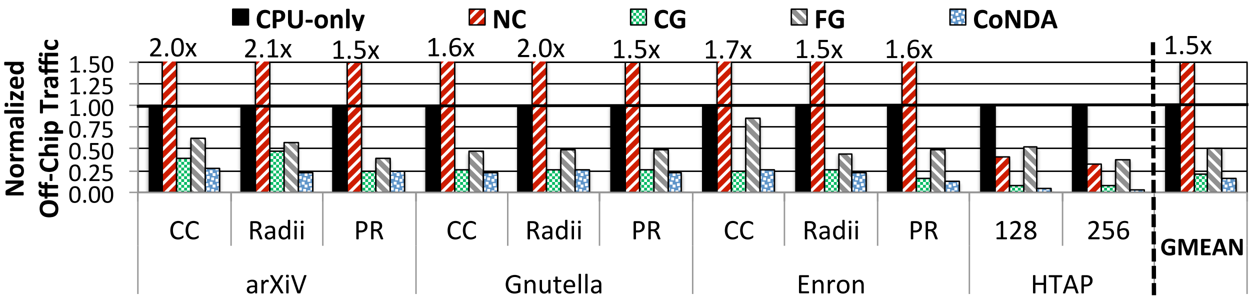


Figure 5.7. Normalized off-chip data movement.

5.6.2. Performance

Figure 5.8 shows the performance improvement for our 16-core system. We make four observations from the figure. First, with no coherence overhead, Ideal-NDA shows that there is significant potential for speedup (on average 1.84x) in our applications when we use NDAs. Second, we observe that CG and NC experience drastic performance losses compared to Ideal-NDA, due to the high costs they have in maintaining coherence. Both CG and NC lead to on average 0.4% and 6.0% performance loss over CPU-only. Third, while FG provides reasonable performance improvements, it still falls far short of Ideal-NDA, achieving only 44.9% of Ideal-NDA's performance benefits. Fourth, unlike the other mechanisms, CoNDA's efficient approach to coherence allows it to retain most of the performance benefits of Ideal-NDA, coming within 10.4% on average. CoNDA improves performance over CPU-only by 66.0%, and over the best previous mechanism, FG, by 19.6%.

We also evaluate the effect of offloading the entire application to NDAs. Running an entire application on NDAs eliminates the need for coherence between the CPU and NDAs. However, our analysis shows that executing these applications entirely on NDAs hurts performance significantly, eliminating on average 82.2% of Ideal-NDA's performance improvement. CoNDA performs 51.7% better than NDA-only, and NDA-only performs only 8.7% better than CPU-only. The reason is that the NDA cannot afford to incorporate complicated logic and large caches due to power and area constraints [42, 77, 99], which, for NDA-only, significantly slows down the parts of the application that are better suited for CPU execution.

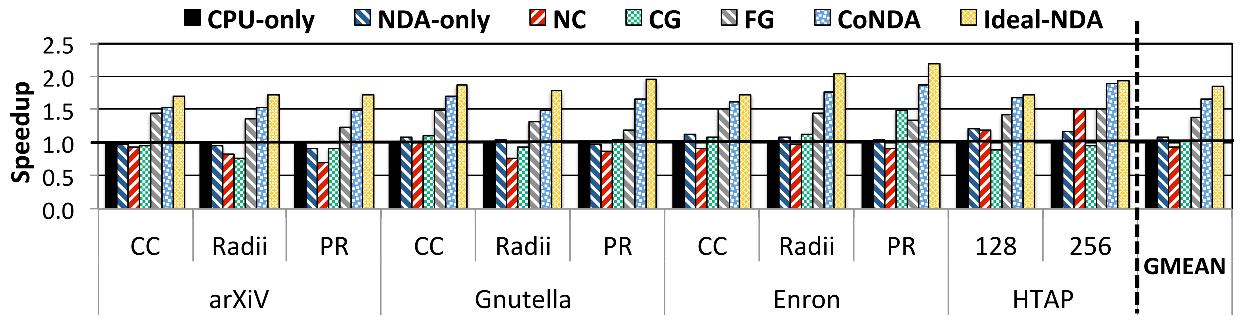


Figure 5.8. Speedup across existing coherence mechanisms.

CoNDA's execution time consists of three major parts: (1) NDA kernel execution, (2) coherence resolution overhead, and (3) re-execution overhead. Our analysis shows that for our applications, the coherence resolution and re-execution overheads take 3.3% and 8.4% of the entire execution time, respectively.

Coherence Resolution Overhead. We find that the coherence resolution overhead is low because (1) CPU threads do *not* stall during resolution unless they access the NDA data region; (2) the `NDAWriteSet` typically contains only a small number of addresses (6 on average), which limits the number of CPU-side invalidations and NDA writebacks (see Section 5.4.6); and (3) resolution mainly involves sending signatures and checking for any necessary coherence operations, which altogether take less than 50 cycles.

Re-Execution Overhead. We find that the overhead of re-execution is small for two reasons. First, as we mention in Section 5.2.2, the collision rate (i.e., the fraction of commit attempts that require re-execution) is low (13.4% on average) for our applications, limiting the number of times an NDA must re-execute and the number of cache lines that must be flushed to DRAM. Second, the re-execution of the NDA kernel portion is significantly *faster* than its original execution. This is because the majority of data and instructions needed by the NDA during re-execution are already in the NDA L1 cache from the previous execution, and CoNDA places a copy of each cache line flushed from the CPU during coherence resolution into the NDA L1 cache.

5.6.3. Memory System Energy

Figure 5.9 shows the memory system energy consumption of the NDA coherence mechanisms. We observe that CoNDA reduces energy consumption over all prior mechanisms, by 18.0% on average over the best mechanism for memory system energy (CG). CoNDA achieves nearly all of the energy reduction potential of Ideal-NDA, coming within 4.4%, and reduces energy consumption by 43.7% over CPU-only. This is because CoNDA successfully reduces off-chip traffic, and eliminates much of the unnecessary coherence traffic of existing coherence mechanisms.

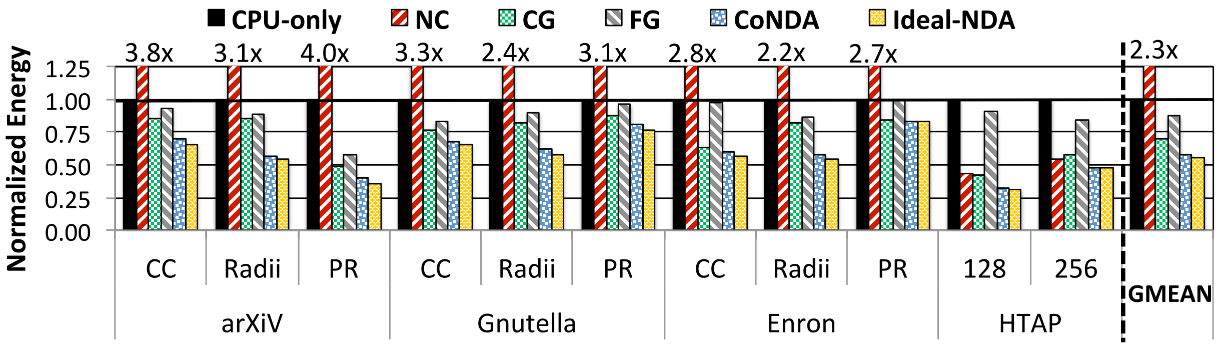


Figure 5.9. Normalized memory system energy.

FG, CG, and NC each introduce coherence overheads that undermine their potential for energy savings. FG exchanges a very high number of off-chip coherence messages between the CPU and the NDAs. While CG reduces memory system energy over CPU-only, it induces a large number of writebacks, which increase DRAM energy consumption by 18.9% CPU-only. The additional DRAM energy cancels out some of the off-chip interconnect energy savings (a 49.1% reduction over CPU-only) that CG provides. We find that CG is more energy efficient than FG because it generates less off-chip coherence traffic. CG performs writebacks only at the beginning of NDA kernel execution while FG generates off-chip coherence traffic on every single coherence miss. NC forces all CPU accesses to the NDA data region to bypass the CPU caches and go to DRAM, which increases the interconnect and DRAM energy over CPU-only by 3.1x and 4.5x, respectively.

5.6.4. Multiple Memory Stacks

In systems that need a large main memory, there can be multiple memory stacks each with NDAs. We evaluate how CoNDA's benefits scale as multiple stacks require coherence, using a representative workload (*PageRank* with the *arXiv* input graph) as a case study. In this study, we assume that there are four CPU cores and four NDAs per stack, and that stacks are connected together using the *processor-centric topology* [209]. We use the local NDA directory in each NDA stack (Section 5.5) to maintain coherence between different stacks (i.e., similar to the distributed directory in NUMA machines [10, 160]) using the MESI coherence protocol.

Off-Chip Traffic. Figure 5.10 shows the normalized off-chip traffic for each mechanism as the number of stacks increases. The NDA and CPU core count increases with stack count, and so does the off-chip memory traffic. Unlike existing NDA coherence mechanisms, off-chip traffic under CoNDA scales well and remains significantly lower than CPU-only as we increase the stack count. FG and CG do *not* scale as well as CoNDA. For FG, increasing the stack count leads to a significantly larger number of coherence misses, and thus, generates more off-chip coherence messages. For CG, the number of writebacks required scales *superlinearly* with stack count (6.2x from 1 to 4 stacks; not shown). NC generates more traffic than CPU-only across all stack counts. In contrast, even with 4 stacks, CoNDA reduces off-chip traffic by 82% over CPU-only, again significantly reducing the cost of NDA coherence.

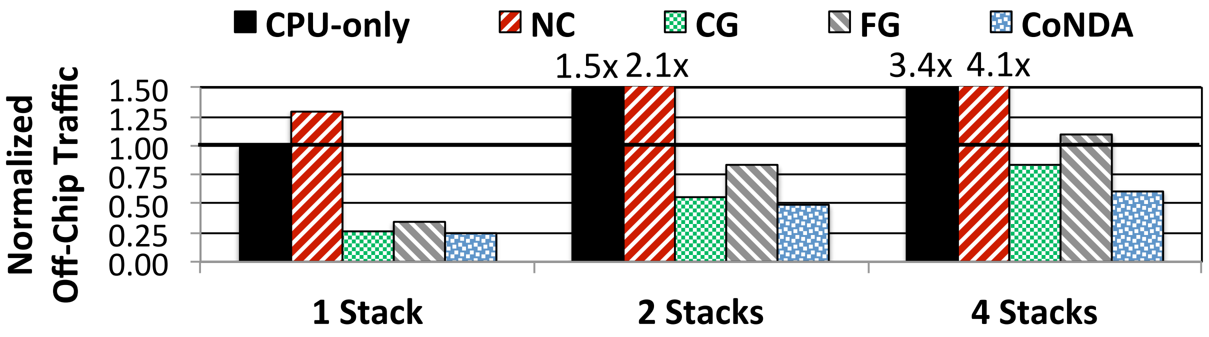


Figure 5.10. Effect of stack count on data movement (lower is better) for *PageRank* with *arXiv* input graph.

Performance. Figure 5.11 shows the normalized speedup for each mechanism as the number of stacks increases. Across all stack counts, we find that CoNDA consistently outperforms CPU-only, FG, CG, and NC. Compared to FG, the best existing mechanism at 4 stacks, CoNDA improves performance by 21.5%. This is because there is more off-chip traffic at higher stack counts, which CoNDA reduces, as we see in Figure 5.10. Unlike CoNDA, as the stack count increases, CG (at 4 stacks) and NC (at 2 and 4 stacks) actually perform *worse* than CPU-only. CG performs worse for two reasons: (1) the high number of flushes; and (2) the increased probability of blocking CPU threads during concurrent execution, with the threads stalled for up to 73.1% of the execution time. FG still outperforms CPU-only as the stack count increases, but its high amount of off-chip coherence requests prevents it from coming close to CoNDA's performance.

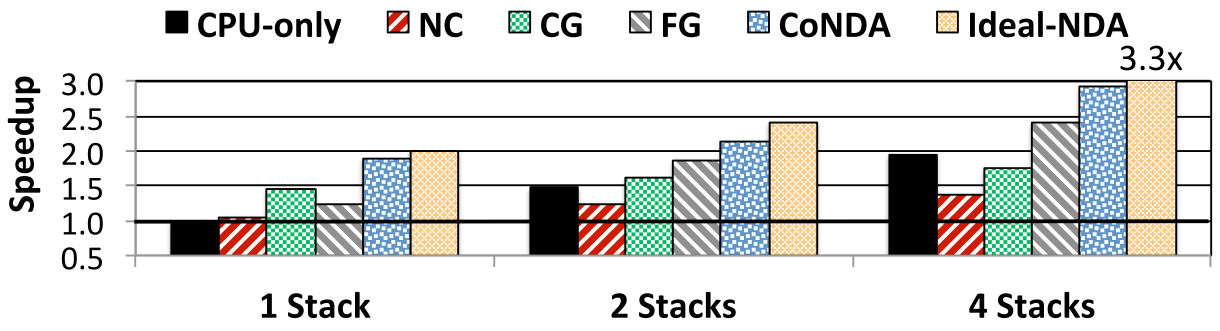


Figure 5.11. Effect of stack count on speedup for *PageRank* with *arXiv* input graph.

5.6.5. Effect of Larger Data Sets

Increasing the data set size significantly increases the benefits of CoNDA. This is because the larger data sets result in a significantly larger number of cache misses than the smaller data sets. As a result, more coherence traffic is generated, which, thus, provides more opportunities for CoNDA to eliminate overheads. To demonstrate this, we evaluate three of our applications with larger data sets: (1) *Connected Components* and (2) *Radii*, and (3) *HTAP-1024*. Figure 5.12 shows the performance improvement normalized to the CPU-only baseline. We find that Ideal-NDA outperforms CPU-only by 9.2x, averaged across these three workloads. CoNDA retains most of the performance benefits of Ideal-NDA, coming within 10.2% on average, and improves performance by 8.4x over CPU-only, 7.7x over NDA-only, and 38.3% over FG (the best prior coherence mechanism). Similar to our observation for the modest size data sets, executing these applications entirely on NDAs (shown by NDA-only) hurts performance significantly, eliminating on average 88.7% of Ideal-NDA's performance improvement. We conclude that as we scale to larger data sets, CoNDA retains its effectiveness at eliminating unnecessary coherence traffic and preserves almost all of the benefits of NDA execution.

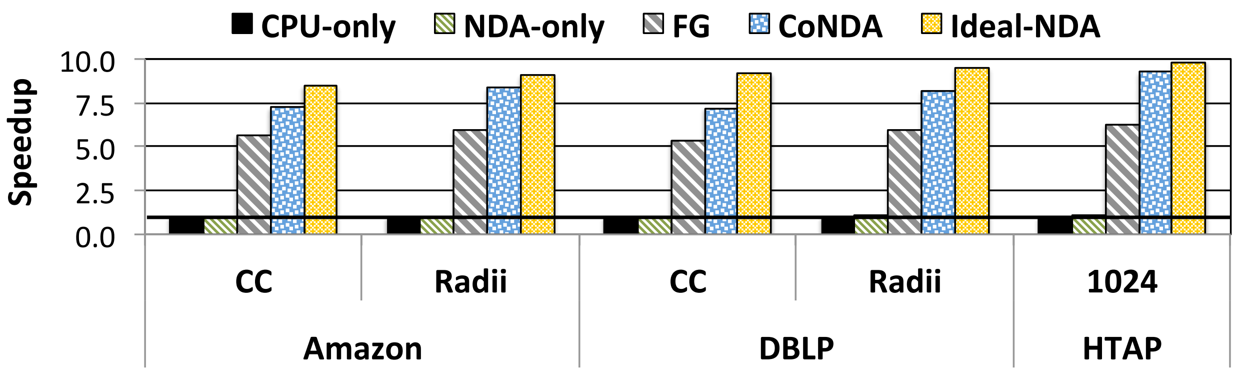


Figure 5.12. Speedup for larger data sets.

5.6.6. Effect of Optimistic Execution Duration

As we discuss in Section 5.4.3, the duration of optimistic execution is determined by the number of addresses that can be held in the signatures without exceeding a target false positive rate, for a fixed signature length. Figure 5.13 shows how varying the number of addresses affects execution time and off-chip traffic (normalized to CPU-only) for two representative workloads: *Connected Components* with the *Enron* input graph, and *HTAP-128*. We make two observations from the figure. First, as the duration increases from 150 addresses to 350 addresses, the total execution time *increases* by 5.7% and 10.5% for *Connected Components* and *HTAP-128*, respectively. This is because when the duration is longer, there are more opportunities for the CPU and NDA to collide on the same cache line. As a result, the conflict rate (i.e., the fraction of coherence resolution attempts that require re-execution) increases by 18.3% and 41.2%, respectively. Second, as the duration increases, the off-chip traffic *decreases*, by 25.7% and 15.5%, respectively. Every time CoNDA performs coherence resolution, the NDA sends its signatures to the CPU across the off-chip interconnect, and a longer duration requires less frequent coherence resolution.

Given our signature size (256 B) and target false positive rate (20%), we conclude that a duration of 250 addresses strikes a good balance between execution time and off-chip traffic.

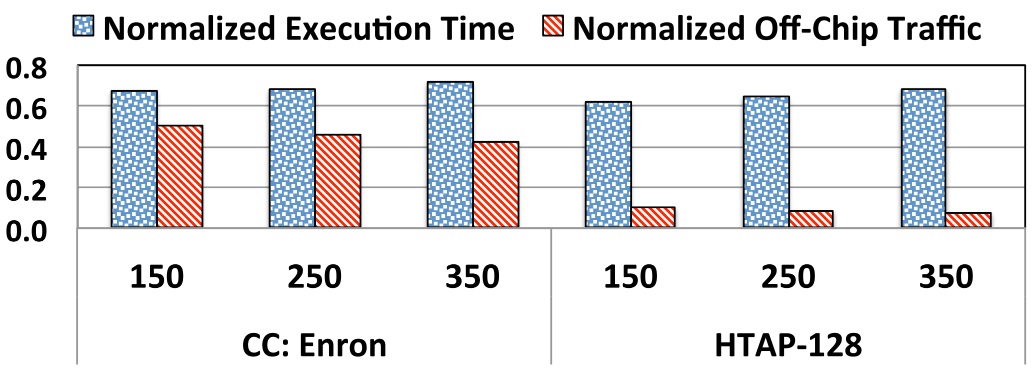


Figure 5.13. Effect of optimistic execution duration.

5.6.7. Effect of Signature Size

We repeat the study from Section 5.6.6, but this time hold the duration of optimistic execution constant at 250 addresses, and instead vary the size of each signature. Figure 5.14 shows how the signature size affects execution time and off-chip traffic (normalized to CPU-only). We make two observations from the figure. First, when the signature size increases from 256 B to 1 kB, the execution time *decreases* by 10.1% for *Connected Components* and 10.9% for *HTAP-128*. Increasing the signature size has a similar effect to decreasing the duration of optimistic execution. In this case, the conflict rate decreases mainly because the false positive rate is lower, by 31.4% and 40.5% for the respective workloads. Second, when the signature size increases from 256 B to 512 B and to 1 kB, the off-chip traffic *increases*, by 32.7% and 31.4%, respectively.

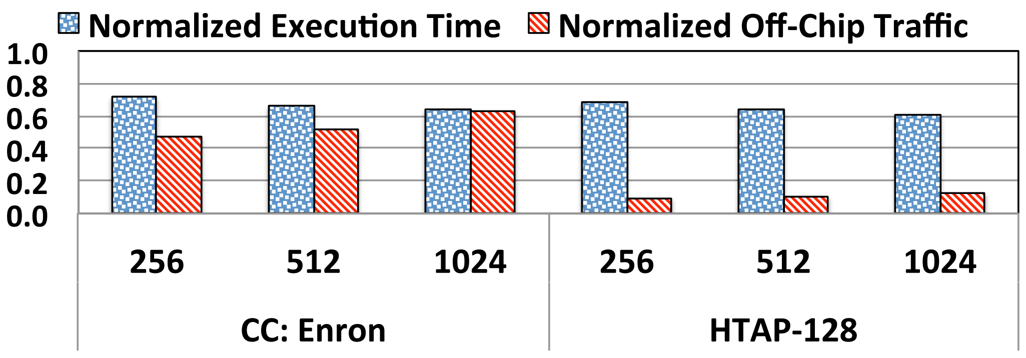


Figure 5.14. Effect of signature size.

We conclude that for a 250-address duration, using a smaller 256 B signature strikes a good balance between storage overhead, execution time, and off-chip traffic across all of our workloads.

5.6.8. Effect of Data Sharing Characteristics

The amount of data shared by CPU threads and NDA kernels, and the rate at which these two collide on the same cache lines during concurrent execution, are intrinsic properties of the application and how it is partitioned. We can group applications into three categories: (1) limited sharing; (2) high amount of sharing, infrequent collisions; (3) high amount of sharing, frequent collisions. So far, our work has shown that several important applications fit into Category 2, and CoNDA is much more effective for these applications than existing coherence mechanisms. In this section, we explore how CoNDA performs for applications in the *other two* categories (1 & 3).

Many NDA applications fall under Category 1, and prior works on NDA often assume this behavior (e.g., [77,93,99,206,302,380]). To show how CoNDA compares to CG, NC, and FG for such applications, we construct two representative benchmarks inspired by prior works [16,42,320]: (1) a matrix tiling operation that is offloaded to an NDA, and (2) a kernel where the NDA performs `memcpy()` and `memset()` on a large region of memory. During NDA execution, the CPU is idle. For these benchmarks, we find that CG, NC, and CoNDA all perform comparably, outperforming CPU-only by 1.83x, 1.85x, and 1.82x, respectively (not shown). All three achieve near-ideal performance, with CoNDA coming within 2.8%. In contrast, FG performs relatively poorly, only coming within 21.1% of Ideal-NDA, due to its high coherence traffic. Even though CG and NC especially cater to such applications, CoNDA still performs competitively, as *very few* rollbacks occur when sharing is limited. Unlike CG and NC, CoNDA provides coherence at a fine granularity (i.e., per cache line) for these applications, making the programming model simpler.

Unlike applications in Categories 1 and 2, applications in Category 3 may not benefit significantly from the concurrent execution of CPU threads and NDA kernels. For such applications, most coherence messages are necessary, and the system would have to spend a large fraction of the total execution time on performing these requests. We develop a representative microbenchmark for this category, where the NDA performs a tiling operation on a matrix while the CPU threads concurrently update and read from the matrix. FG, CG, and NC all hurt the performance of the benchmark, performing 33.4% worse (averaged across the three coherence mechanisms) than CPU-only (not shown) for the same reasons that we discuss in Section 5.2.3. While CoNDA does significantly better, performing only 2.1% worse than CPU-only, it still falls significantly short of Ideal-NDA, by 51.2%, due to frequent rollbacks. We note, however, that applications where a large portion of the execution is done on an accelerator [91, 178, 182, 196, 217, 252, 306] rarely exhibit this kind of behavior where the CPU and the accelerator collide frequently on shared data.

We conclude that for the vast majority of accelerator-centric applications, we expect that CoNDA (1) either significantly outperforms or performs competitively with existing NDA coherence mechanisms, and (2) achieves near-ideal performance.

5.7. Summary

Many applications can harness near-data accelerators (NDAs) to gain significant performance and energy benefits. However, enforcing coherence between an NDA and the rest of the system is a major challenge. We extensively analyze NDA applications and existing coherence mechanisms, and observe that (1) a majority of off-chip coherence traffic is unnecessary, and (2) a significant portion of off-chip data movement can be eliminated if a coherence mechanism has insight into NDA memory accesses. Based on our observations, we propose CoNDA, a coherence mechanism that lets an NDA optimistically execute code assuming that it has coherence permissions. Optimistic execution enables CoNDA to gather information on memory accesses, and exploit the information to minimize unnecessary off-chip data movement for coherence. Our results show that CoNDA improves performance and reduces energy consumption compared to existing coherence mechanisms, and comes close to the energy and performance of a no-cost ideal coherence mechanism. We conclude that CoNDA is an effective coherence mechanism for NDAs, and hope that this work encourages the development of other mechanisms for NDA coherence.

Chapter 6

Mitigating Edge Machine Learning Inference Bottlenecks: An Empirical Study on Accelerating Google Edge Models

Modern consumer devices make widespread use of machine learning (ML). The growing complexity of these devices, combined with increasing demand for privacy, connectivity, and real-time responses, has spurred significant interest in pushing ML inference computation to the edge (i.e., into these devices, instead of in the cloud) [57, 376, 383]. Due to the resource-constrained nature of edge platforms, they now employ specialized energy-efficient accelerators for on-device inference (e.g., Google Edge TPU [114], NVIDIA Jetson [282], Intel Movidius [179]). At the same time, neural network (NN) algorithms are evolving rapidly, which has led to a myriad of NN models (e.g., CNNs [339], LSTMs [315], GRUs [67], Transducers [135, 158], hybrid models [74, 367]), each targeting various applications (e.g., face detection [339], speech recognition [158, 240, 315], translation [379], image captioning [74, 367]). Edge ML accelerators are designed to provide a one-size-fits-all solution across a wide variety of NN models, while being mindful of edge device area and energy constraints.

Unfortunately, the one-size-fits-all approach has made it very challenging for edge ML accelerators to execute all of these models in a way that simultaneously achieves high energy efficiency

(TFLOP/J), computational throughput (TFLOP/s), and area efficiency (TFLOP/mm^2) for each workload. Our goal is to revisit the design of edge ML accelerators such that they are aware of and can exploit the layer variation within and across edge NN models, using a hardware-software co-design approach aware of PIM.

6.1. Summary of Key Insights and Observations

We conduct an in-depth analysis of inference execution on a commercial Edge TPU [114], across 24 state-of-the-art Google edge models spanning four popular NN model types: (1) CNNs, (2) LSTMs [315], (3) Transducers [158, 240, 305], and (4) RCNNs [74, 367]. These models are used in several Google mobile applications, such as image classification, object detection, semantic segmentation, automatic speech recognition, and image captioning. Based on our analysis (Section 6.2), we find that the accelerator suffers from three major shortcomings. First, the accelerator utilizes only 1/4 of its peak throughput, averaged across all models (and less than 1% utilization for LSTMs and Transducers, the worst case). Second, despite using specialized logic, the accelerator provides only 37% of its theoretical peak energy efficiency (TFLOP/J) on average. Third, the accelerator's memory system is often a large bottleneck. As an example, while large on-chip storage buffers (e.g., several megabytes) account for a significant portion of energy consumption (e.g., 48.1% static and 36.5% dynamic energy during CNN inference), they are often ineffective in reducing off-chip accesses, and cannot accommodate the parameters of larger NN models.

To identify the root cause of these shortcomings, we perform the first comprehensive per-layer analysis of the google edge NN models, and make two key observations. First, there is significant variation in terms of layer type, shape, and characteristics (e.g., FLOP/Byte ratio, footprint, intra- and inter-layer dependencies) *across* the models. For example, Transducer layers differ drastically (by as much as two orders of magnitude) from CNN layers in terms of parameter footprint and FLOP/B. Second, even *within each model*, there is high variation in terms of layer types and shapes (e.g., pointwise, depthwise, fully-connected, standard convolution, recurrent). This leads to up to two orders of magnitude of variation for layer characteristics within a single model. We quantify for the first time how intra-model variation is dramatically higher in edge models compared to previously-studied traditional models (e.g., [220, 339]) because edge models employ several techniques (e.g., separable convolutions [165, 175]) to reduce computational complexity and layer footprint, in order to optimize the models for resource-constrained edge devices.

While our analysis uncovers the significant heterogeneity that exists in edge NN layers, we find that the key components of edge ML accelerators, including the processing element (PE) array, dataflow, memory system (on-chip buffers and off-chip memory), and on-chip network, are completely oblivious to this heterogeneity, which leads to the three accelerator shortcomings that we observe in Section 6.2. Instead, state-of-the-art edge ML accelerators [58, 95, 100, 179, 181, 282] take a monolithic design approach, where they equip the accelerator with a large PE array, large on-chip buffers, and a fixed dataflow (e.g., output stationary). While this approach might work for a specific group of layers (e.g., traditional convolutional layers with high compute intensity and high data reuse), we find that it fails to achieve the desired energy efficiency, throughput, and area efficiency across the significantly more diverse state-of-the-art edge NN models. One such example is the memory system, where state-of-the-art accelerators employ highly overprovisioned on-chip buffers that are unable to effectively reduce off-chip parameter traffic and provide high bandwidth to PEs, leading to PE underutilization. Another example is the fixed dataflow used by the accelerators across all layers. Due to the drastic variation across different layers, the fixed dataflow often misses spatial/temporal reuse opportunities across layers.

A number of recent works [57, 225] cater to NN variation by enabling reconfigurability for parts of the accelerator. For example, Eyeriss v2 [57] provides the ability to reconfigure the on-chip interconnect and make use of a smaller PE array. Unfortunately, as models become more diverse and go beyond the structure of more traditional CNNs, reconfigurable accelerators face three issues: (1) they do not provide the ability to reconfigure a number of essential design parameters (e.g., on-chip buffers, memory bandwidth); (2) they can require frequent online reconfiguration to cater to increasing intra-model heterogeneity, with associated overheads; and (3) they make it difficult to co-design the dataflow with key components such as the memory system. The *key takeaway* from our extensive analysis of Google edge NN models on the Edge TPU is that *all key components* of an edge accelerator (i.e., PE array, dataflow, memory system) must be co-designed based on specific layer characteristics to achieve high utilization and energy efficiency. Our goal is to revisit the design of edge ML accelerators such that they are aware of and can fully exploit the growing variation within and across edge NN models. Our goal is to revisit the design of edge ML accelerators such that they are aware of and can fully exploit the growing variation within and across edge NN models.

To this end, we propose Mensa, the first general HW/SW composable framework for ML acceleration in edge devices. The overarching key idea of Mensa is to incorporate multiple heterogeneous accelerators, each of which is small and tailored to the characteristics of a particular subset of layers, and to use a runtime scheduler that assigns each layer to the accelerator that is best suited to the layer's characteristics and inter-layer dependencies. As we study the characteristics of different layers in our edge models, we make an important discovery: the layers naturally group into a small number of clusters. In each cluster, the layers belonging to that cluster have at least one common characteristic that differs significantly from the layers in other clusters, and that strongly influences the best hardware design for those layers. As a result, the number of different accelerators that need to be implemented using Mensa is limited to the number of clusters (five for our Google edge models), allowing an implementation of Mensa to fit within the tight constraints of edge devices.

Using our insight about layer clustering, we develop one possible design for Mensa that is optimized for our Google edge workloads. We identify the key characteristics that lead to the formation of each cluster, and find that our accelerator designs should center around two of them (memory boundedness, and activation/parameter reuse opportunities) that have the greatest impact on hardware efficiency. First, we need separate accelerators for compute-centric clusters and for data-centric clusters. Layers in compute-centric clusters make heavy use of their multiply-accumulators (MACs), have small parameter footprints, and high parameter reuse, requiring only small on-chip buffers and limited memory bandwidth. In contrast, layers in data-centric clusters do not use MACs heavily, and have high memory bandwidth requirements, requiring very different hardware resources from compute-centric layers. Second, we need separate accelerators to account for different types of dataflow across the clusters. We find that an accelerator's dataflow depends on the amount and types of data reuse that a layer can make use of, along with whether or not inter- and/or intra-layer dependencies are present. The choice of dataflow strongly influences the entire design of each accelerator. Notably, we find that multiple clusters share common reuse and dependency behavior. Using this, we consolidate our five clusters of layers down to three distinct types of layer behavior that our implementation needs to account for.

As a result, we design three accelerators for our implementation: Pascal (for compute-centric layers), Pavlov (for LSTM-like data-centric layers), and Jacquard (for other data-centric layers). We employ a template-based design approach, customizing each accelerator based on layer characteristics while maintaining the tiled architecture of the baseline accelerator. While this limits the degree of customization, it allows Mensa to seamlessly run models compiled using existing highly-optimized toolchains (e.g., Edge TPU compiler [115]) and minimizes programmer burden. Pascal uses a dataflow that enables temporal reduction of output activations and spatial multicasting of parameters. This dataflow allows us to design a memory system with an on-chip buffer that is 16x smaller than our baseline state-of-the-art accelerator, and greatly reduces on-chip network traffic, while still keeping the processing elements (PEs) highly utilized. Pavlov uses a dataflow that enables temporal reduction of output activations, and identifies opportunities to schedule the parallel execution of layer operations in a way that improves parameter reuse, reducing off-chip parameter traffic. Jacquard uses a dataflow that exposes reuse opportunities in parameters and reduces the size of the parameter buffer. To take advantage of the data-centric nature of both Pavlov and Jacquard, we implement both accelerators in the logic layer of 3D-stacked memory to provide high bandwidth and mitigate data movement costs, and use significantly smaller PE arrays for the accelerators compared to the PE array in Pascal.

Our evaluation shows that compared to our baseline Edge TPU, Mensa reduces total inference energy by 66.0%, improves energy efficiency (TFLOP/J) by 3.0x, and increases computational throughput (TFLOP/s) by 3.1x (averaged across all 24 Google edge NN models). Mensa improves inference energy efficiency and throughput by 2.4x and 4.3x over Eyeriss v2, a state-of-the-art accelerator.

6.2. Analysis

We analyze 24 Google edge models (including CNNs, LSTMs, Transducers, and RCNNs) using a commercial Edge TPU as our baseline accelerator. The Edge TPU has a generic tiled architecture, similar to other state-of-the-art accelerators [58, 99, 100, 191]. It includes a 2D array of PEs (64x64), where each PE has a small register file to hold intermediate results. The accelerator has two large SRAM-based on-chip buffers to hold model parameters and activations [115].

6.2.1. Google Edge TPU Shortcomings

We analyze inference execution on the accelerator using 24 edge NN models that serve as internal benchmarks for the Google Edge TPU, (with CNNs, LSTMs, Transducers, and RCNNs), and find three major shortcomings:

1. The accelerator often suffers from extreme underutilization of the PEs. The Edge TPU has a theoretical peak throughput of 2 TFLOP/s, which is orders of magnitude higher than CPUs and GPUs. However, we find that the accelerator operates *much* lower than peak throughput during inference execution (75.6% lower on average). Figure 6.1 (left) shows the roofline model of throughput for the Edge TPU, along with the measured throughput of all of our edge models. The PE utilization is consistently low across all models. Transducer and LSTM models have the greatest underutilization, with both achieving less than 1% of peak throughput. While CNN and RCNN models do somewhat better, they achieve only 40.7% of peak utilization on average (going as low as 10.2%).

2. Despite using specialized logic, the Edge TPU operates far below its theoretical maximum energy efficiency. We use a similar approach to prior work [62] to obtain a roofline for energy efficiency. Figure 6.1 (right) shows the energy roofline for Edge TPU, along with the efficiency achieved with each model. Note that unlike the throughput roofline model, the energy roofline is a smooth curve because we cannot hide memory energy (as opposed to memory transfer time, which can be overlapped with computation time and results in the sharp knee for the throughput roofline). We find that across all models, the accelerator fails to get close to the maximum energy efficiency,

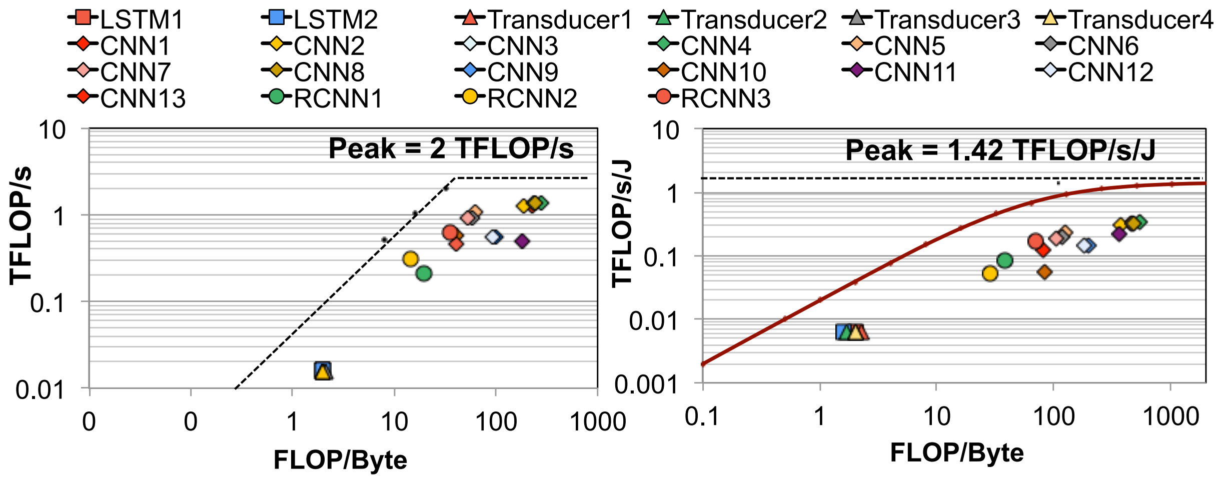


Figure 6.1. Throughput roofline (left) and energy roofline (right) for our baseline accelerator across all models.

falling 62.4% lower on average. The energy efficiency is particularly low (33.8% on average) for LSTM and Transducer models, but even the best CNN model achieves only 50.7% of the maximum efficiency.

3. The accelerator's memory system design is neither effective nor efficient. Figure 6.2 shows the energy breakdown during inference execution across different models. We make three key observations from this figure. First, on-chip buffers account for a significant portion of both static and dynamic energy, mostly because of their large size . For example, for CNN models, 48.1% of the static energy and 36.5% of the dynamic energy is spent on accessing and storing parameters in the on-chip buffers. Second, while the buffers consume a significant amount of area (79.4% of the total accelerator area) and energy, they often do not reduce off-chip memory accesses.

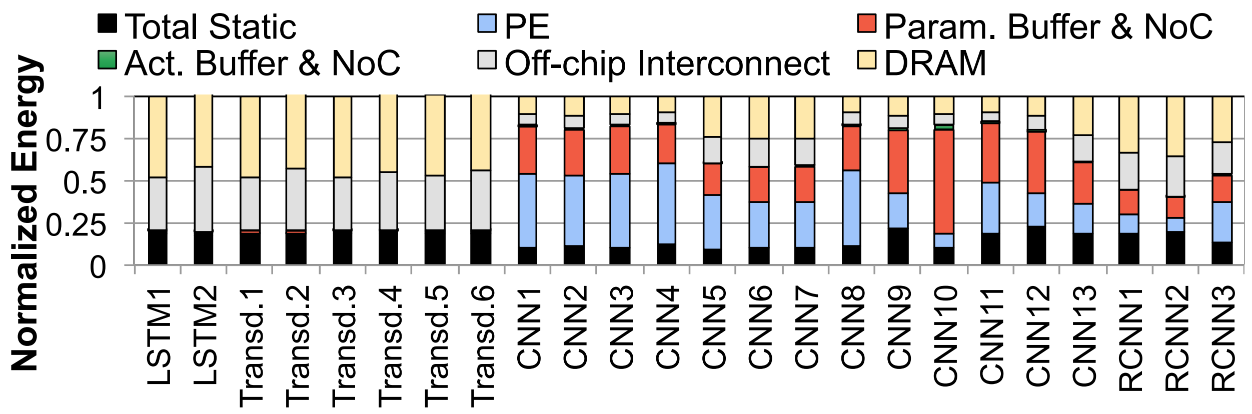


Figure 6.2. Energy breakdown during inference execution across all models.

For Transducers and LSTMs, only 11.9% of the model parameters can fit into the buffer, and for CNNs the on-chip buffer can accommodate only 15.0% of the parameters on average. Increasing the buffer size does not help: if we increase the buffer size to 8x, this provides a latency and energy reduction of only 37.6% and 40.3%, respectively, for Transducers. This is because (1) even such a large buffer can cache only 46.5% of the parameters; and (2) the leakage and access energy increase significantly with a larger buffer size, offsetting other benefits of a larger capacity. Third, both on-chip (i.e., distributing parameters from the on-chip buffer to the PEs) and off-chip (i.e., accesses to DRAM) network traffic for model parameters results in high energy and performance costs. On average, 50.3% of the total Edge TPU energy is spent on off-chip parameter traffic (DRAM accesses and off-chip interconnect), and 30.9% of the total energy is spent on on-chip parameter traffic. We conclude that the Edge TPU's memory system is highly inefficient.

6.2.2. Layer-Level Study of Google Edge Models

To understand where the Edge TPU's [114] pitfalls come from, we analyze the models themselves in significant detail.

6.2.2.1. Analysis of LSTMs and Transducers

We identify three key properties of LSTMs and Transducers in our edge model analysis.

1. Large parameter footprint. Each gate in an LSTM cell has an average of 2.1 million parameters, which includes parameters for both input (W_x) and hidden (W_h) matrices (as shown in Figure 6.3, left). The large parameter footprint of LSTM gates results in large footprints for LSTM layers (up to 70 million parameters), and in turn, LSTM and Transducer models that include such layers. Figure 6.3 (right) shows the total footprint vs. the FLOP/B ratio (which indicates arithmetic intensity) across the layers of representative CNNs, LSTMs, and Transducers (the trend is the same across all models). We observe from the figure that that layers from LSTMs and Transducers have significantly larger footprints (with an average footprint of 33.4 MB) than layers from CNNs.

2. No data reuse and low computational complexity. For these layers, the FLOP/B of both W_x and W_h is one, as shown in Figure 6.3 (right). The accelerator fetches W_x and W_h for each

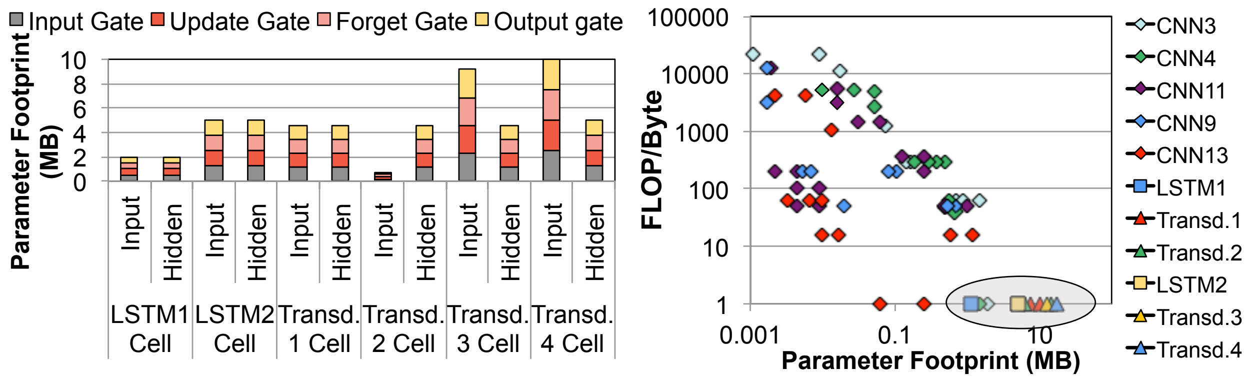


Figure 6.3. Parameter footprint of W_x and W_h for different LSTM gates for LSTMs and Transducers (left), and layer parameter footprint vs. FLOP/B (right).

LSTM gate from DRAM, accesses them once to perform the input and hidden MVMs, and then does not touch the parameters again until the next LSTM cell computation, resulting in no reuse. Furthermore, these models have much a lower computational complexity than CNNs and RCNNs, with 67% fewer MAC operations on average.

3. Intra- and inter-cell dependencies. Two types of dependencies exist within LSTM layers, both of which affect how the accelerator schedules gate computations: (1) *inter-cell dependencies*, which are due to recurrent connections (h_t) across cells; and (2) *intra-cell dependencies*, which enforce the following order between gates: all four gate computation need to be done before the cell state (c_t) gets updated, after which the hidden vector (h_t)) is generated. To respect these dependencies, the accelerator schedules cells computation in a sequential manner (similar to state-of-the-art accelerators [95, 191]). This means that cell computation cannot start until h_{t-1} is generated by the previous cell. For each LSTM cell, the accelerator treats each gate as two fully-connected (FC) layers (corresponding to input MVM and hidden MVM), and runs the gates sequentially.

However, we find that this scheduling is not efficient. Due to the sequential execution of FC layers, the latency of updating c_t and generating h_t become significantly higher, which in turn degrades PE utilization (as the PEs spend more time waiting for cell calculation to be completed). We find that, despite these dependencies, there are still multiple ways to parallelize computations within LSTM cells and improve PE utilization. For example, gates within each cell can be scheduled in any order, since there are no dependencies between the gates, and the input and hidden MVMs within each gate are independent. Since the scheduler in the baseline accelerator simply treats each gate computation as an FC layer (similar to the FC layer in CNN models), it is oblivious of these opportunities for parallelization.

6.2.2.2. Analysis of CNNs

Our analysis of edge CNN models reveals two interesting insights. First, we find that edge CNN models unlike layers in traditional CNNs (e.g., AlexNet [220], VGG [339]), which tend to be relatively homogenous, we find that the layers in edge CNN models exhibit significant heterogeneity in terms of type (e.g., depthwise, pointwise), shape, and size. We find that this is often because these models employ several decomposition techniques [165, 175] to reduce the computational complexity and footprint of layers, in order to make them more friendly for the constrained edge devices. As an example of layer diversity, Figures 6.4 and 6.5 show the number of MAC operations and parameter footprint across different layers for four CNN models. We find that the MAC intensity and parameter footprint vary by a factor of 200x and 20x across different layers.

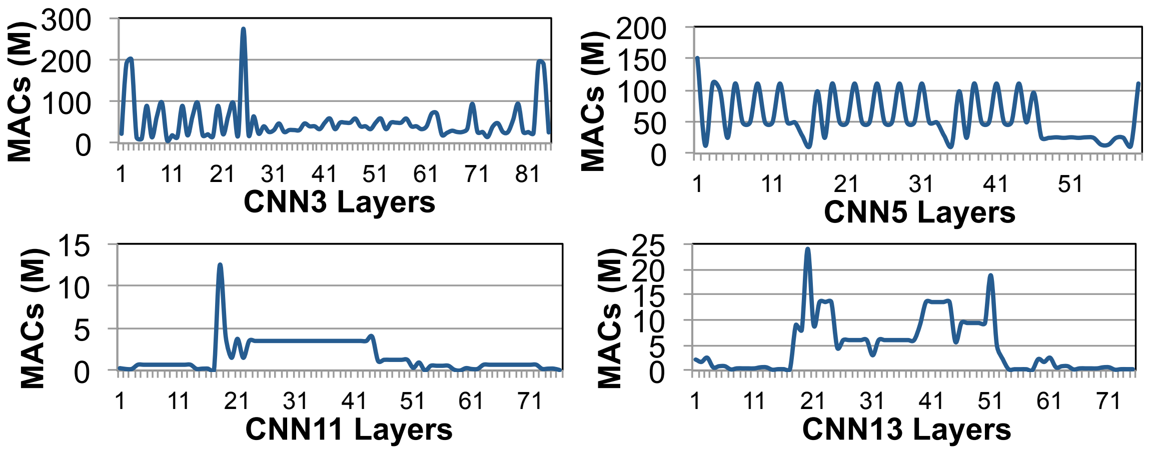


Figure 6.4. Analysis of number of MAC operation across different layers for four CNN models.

Second, we find that layers exhibit significant variation in terms of data reuse patterns for parameters and for input/output activations. For example, a depthwise layer includes only one channel, which leads to very low activation reuse, while a pointwise layer takes in a $1 \times K$ filter (K is the depth of the input channel) and convolves it with an input activation across different input channels, leading to much higher activation reuse. We also observe variation in data reuse across

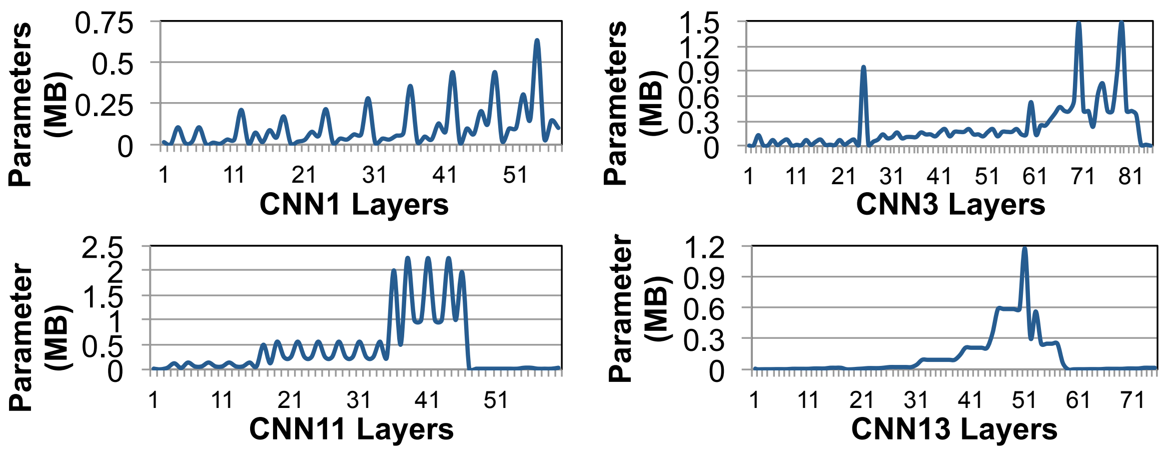


Figure 6.5. Analysis of parameter footprint across different layers for four CNN models.

layers of the same type. For example, initial/early standard convolution layers in edge CNNs have a shallow input/output channel depth, large input activation width/height, and very small kernels, resulting in very high parameter reuse. In comparison, standard convolution layers that are placed toward the end of the network have a deep input and output channel depth, small input activation width/height, and a large number of kernels, resulting in very low parameter reuse. This variation in reuse is illustrated in Figure 6.3 (right), which shows that the FLOP/B ratio varies across different layers for five representative CNN models by a factor of 244x.

6.2.2.3. Analysis of RCNNs

RCNNs include layers from both CNNs and LSTMs. As a result, individual layers from RCNN models exhibit the same characteristics we discussed above for LSTM and CNN layers. We find that layers from RCNNs exhibit on average significantly higher footprints and lower FLOP/B ratios than CNNs, as they include both LSTM and CNN layers. Due to the inclusion of both layer types, we observe significantly more variation across RCNN layers characteristics as well.

6.2.2.4. Sources of Accelerator Pitfalls

PE Underutilization. We identify three reasons why the accelerator falls significantly short of peak throughput. First, while some layers have high parameter reuse (e.g., pointwise layers, with a 1200 FLOP/B ratio), other layers exhibit very low reuse (1–64 FLOP/B) while having large parameter footprints (0.5–5 MB). As we discuss in Section 6.2.2.1, our LSTM layers perform only one FLOP for every parameter byte, often leaving the PEs idle. The large footprint makes the on-chip buffer ineffective at parameter caching, forcing most parameter lookups to go to DRAM. The bandwidth of modern commercial DRAM (e.g., 32 GB/s for LPDDR4 [188]) is two orders of magnitude below the 2 TB/s bandwidth needed to sustain peak PE throughput when only one FLOP/B is performed. Many (but not all) layers end up with a similar bandwidth problem, and PEs spend the majority of time waiting for parameters to arrive from DRAM. Note that the batch size is 1 for edge inference (to meet real-time latency requirements), further exacerbating the problem.

Second, the accelerator does not provide a custom dataflow optimized for each layer. As we identified in Section 6.2.2, layers both across and within models exhibit high variation in terms of data reuse patterns. This variation necessitates the need for *different* dataflows for different layers, where each data flow exposes a different set of reuse opportunities for parameters and activations. However, state-of-the-art edge NN accelerators typically employ a *single* dataflow that is designed for high spatial/temporal reuse [224, 225]. The missed reuse opportunities in many of the model layers causes PEs to needlessly wait on retrieving previously-accessed data that was not properly retained on-chip.

Third, the different shapes and inter-/intra-layer dependencies across different types of layers (e.g., LSTM cell, standard convolution, depthwise, pointwise, fully-connected) makes it challenging to fully utilize a PE array with a fixed size, which is the case in state-of-the-art accelerators [57]. To cater to these differences across layers, there is a need for both better scheduling (e.g., uncovering parallel computation opportunities as we found in Section 6.2.2.1) and appropriately sizing the PE arrays in order to maintain efficient utilization.

Poor Energy Efficiency. We find three major sources of energy inefficiencies. First, the accelerator incurs high static energy costs because (1) it employs a large overprovisioned on-chip buffer, and (2) it underutilizes PEs. Second, the on-chip buffers consume a high amount of dynamic energy, as we saw for CNN layers in Section 6.2.1. Third, the accelerator suffers from the high cost of off-chip parameter traffic. On-chip buffers fail to effectively cache parameters for many layers due to layer diversity, causing 50.3% of the total inference energy to be spent on off-chip parameter traffic.

Memory System Issues. We uncover two sources of memory system challenges. First, due to layer diversity, on-chip buffers are ineffective for a large fraction of layers. As we discuss in Section 6.2.2.1, LSTM gates have large parameter footprints and zero parameter reuse, rendering the on-chip buffer useless for a majority of LSTMs and Transducers, and for a significant fraction of RCNN layers. For CNN layers, we find that those layers with low data reuse account for a significant portion of the entire model parameters (e.g., 64% for CNN6). This means that the on-chip buffer fails to cache a large portion of the parameters for CNN models. As a result, despite being several megabytes in size, the on-chip buffer is effective only for a small fraction of layers, which have an average parameter footprint of only 0.21 MB.

Second, due to having an unnecessarily large on-chip parameter buffer, accesses from those layers with high data reuse become very costly. For those layers, even though they have a small footprint, they perform a large number of buffer accesses. As the on-chip buffer is overprovisioned, the large capacity leads to a high dynamic energy cost per access, even though the capacity is not needed.

6.2.3. Key Takeaways

Our analysis provides three key insights: (1) there is significant variation in terms of layer characteristics *across* and *within* state-of-the-art Google edge models, (2) the monolithic design of the Edge TPU is the root cause of its shortcomings, and (3) to achieve high utilization and energy efficiency, all key components of an edge accelerator (PE array, dataflow, on-chip memory, off-chip memory bandwidth) must be customized based on layer characteristics.

6.3. Mensa Framework

As we observe in Section 6.2, the shortcomings of existing edge ML accelerators come from their monolithic design, which is inefficient across the wide amount of heterogeneity found in the layers of edge NN models. We propose Mensa, a new framework for edge ML acceleration that can successfully harness this heterogeneity at high efficiency.

6.3.1. High-Level Overview

Mensa distributes the layers from an NN model across a collection of smaller hardware accelerators that are carefully specialized towards the properties of different layer types. A drawback of current monolithic accelerators is that by designing for a wide range of layers, many of their hardware resources are overprovisioned or introduce high inefficiency. By specializing each accelerator to a subset of layers, Mensa avoids these issues, resulting in an accelerator with a much smaller area that achieves high throughput and efficiency for those layers. Mensa consists of (1) a collection of heterogeneous hardware accelerators; and (2) a runtime scheduler that determines which accelerator each layer in a model should execute on, using a combination of model and hardware characteristics. As we show in Section 6.4, layers tend to group together into a small number of clusters, allowing designers to typically keep the number of accelerators low.

Mensa is designed as a framework that can support a wide range of architectural implementations. This allows Mensa to be optimized to specific system needs, which is critical to keep resource utilization to a minimum in resource-constrained edge devices, and lets the framework adapt easily to future types of NN models that we expect will arise in the future. We discuss one example implementation of Mensa in Section 6.4, which caters to the Google edge models that we analyze, to illustrate the effectiveness of our framework.

6.3.2. Scheduler

The goal of Mensa's software runtime scheduler is to identify which accelerator each layer in an NN model should run on. Each of the accelerators in Mensa caters to a specific cluster of layers, where the cluster is defined using specific layer characteristics (e.g., type, footprint, data reuse, dependencies). For a given hardware configuration, the scheduler has two pieces of information: (1) the characteristics of each cluster; and (2) which hardware accelerator is best suited for each cluster. Similar to how chipset drivers are configured, this information is generated once during initial setup of a system, and can be modified with an updated driver version to account for new clusters.

When an NN model runs on Mensa, the scheduler generates a mapping between layers and accelerators in Mensa. The scheduler uses the NN model (including a directed acyclic graph that represents communication across model layers) and the configuration information in the driver to determine this mapping. The mapping is generated in two phases.

In the first phase, the scheduler iterates through each layer in the model, and identifies the ideal hardware accelerator for each layer *in isolation* (i.e., without considering communication overhead). The scheduler determines two properties for each layer: (1) the cluster that the layer belongs to; and (2) the target accelerator for the layer; While this phase works to maximize accelerator throughput and energy efficiency for each layer, the resulting schedule may be sub-optimal for efficiency, because it does not consider the overhead of transferring activations or communicating dependencies (e.g., h_t in LSTM cells) between different layers. This can have a large impact on the overall framework performance and energy if the amount of communication is large.

In the second phase, our scheduler accounts for the communication overhead using a simple cost analysis algorithm. For each layer, if the following layer (which we call the *subsequent layer*) is mapped by the first phase to a different hardware accelerator than the current layer, the algorithm calculates the total amount of data that needs to be communicated between the two layers, and uses a simple heuristic to estimate the total cost of (1) using the phase one mapping (i.e., moving data to the subsequent layer's accelerator and running that layer on its optimal accelerator), or (2) moving the subsequent layer (i.e., running the subsequent layer on the current layer's accelerator, which eliminates accelerator-to-accelerator communication, but introduces inefficiencies due to running the subsequent layer on a sub-optimal accelerator). If the cost of moving the subsequent layer is lower than the cost of using the phase one mapping, the subsequent layer is remapped to the current layer's accelerator.

Once the second phase is complete, Mensa begins model execution using the generated mapping. The scheduler ensures that each layer is executed completely in a single accelerator. Thus, there is no need for intra-layer data synchronization. On the other hand, due to the dependencies between layers, different accelerators may need to communicate with each other. Parameters are read-only and are always fetched from DRAM (i.e., no need for data synchronization). However, activations need to be transferred if two consecutive layers run on different accelerators. Mensa uses DRAM to synchronize activations across accelerators, which avoids the need for sophisticated coherence mechanisms between on-chip and near-data accelerators [43].

6.4. Implementing Mensa

We now discuss a Mensa implementation for our Google edge models. We start by identifying layer clustering in these models (Section 6.4.1). Using the unique characteristics for each cluster, we determine which characteristics have the greatest impact on accelerator design, and use that to guide the number of accelerators that we need (Section 6.4.2).

6.4.1. Identifying Layer Clusters

We revisit the models that we analyze in Section 6.2. For each layer, we study the correlation between different characteristics. As an example, Figure 6.6 shows how the parameter footprint and the number of MAC operations correlate to parameter reuse (FLOP/B) for a representative set of layers from five CNNs, two LSTMs, and two Transducers, in order to improve figure clarity. Based on all of the layer characteristics that we analyze, we observe across all layers from all models (not just the representative layers or correlations plotted) that 97% of the layers group into one of five clusters, as indicated in Figure 6.6. We discuss each cluster below, and find that there is at least one identifying characteristic for each cluster that differs significantly from the other clusters.

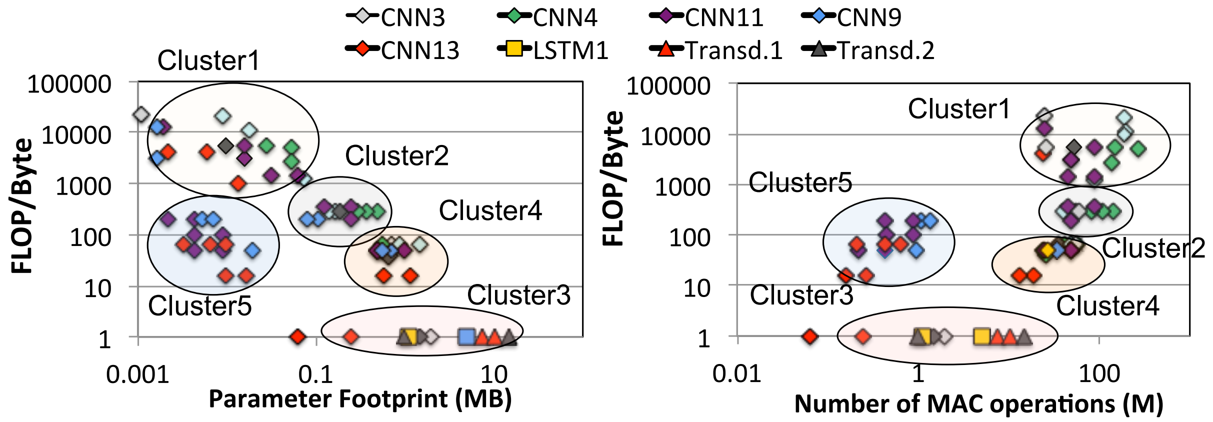


Figure 6.6. Parameter footprint vs. parameter FLOP/B ratio (left) and number of MAC operations (in millions) vs. parameter FLOP/B ratio (right) across layers from representative models.

Cluster 1. Layers in this cluster have (a) a very small parameter footprint (1–100 kB), (b) a very high FLOP/B ratio (780–20K), and (c) high MAC intensity (30M–200M). These layers exhibit high activation footprints and data reuse as well. A majority of layers in this cluster are standard convolution layers with shallow input/output channels and large input activation width/height. We find that these layers are mostly found among early layers in models, and typically achieve high PE utilization (on average 82%).

Cluster 2. Layers in this cluster have (a) a small parameter footprint (100–500 kB; 12x higher on average vs. Cluster 1), (b) a moderate FLOP/B ratio (81–400; up to 10x lower than Cluster 1), and (c) high MAC intensity (20M–100M). They exhibit high activation footprints and activation data reuse as well. Many of the layers belong to pointwise layers, which have high parameter reuse due to convolving 1xK filters (K is input channel depth) with input activation across different channels. Other layers in the cluster include standard convolution layers commonly found in the middle of CNN networks, with deeper input/output channels and smaller activation width/height than the convolution layers in Cluster 1, We find that Cluster 2 layers have lower PE utilization (64%) than layers in Cluster 1.

Cluster 3. Layers in this cluster have (a) a very large parameter footprint (0.9–18 MB), (b) minimal parameter reuse, and (c) low MAC intensity (0.1M–10M). These layers exhibit small activation footprints but rather high activation reuse. The majority of these layers are from LSTM gates in LSTMs and Transducers, or are fully-connected layers from CNNs. These layers have very low PE utilization (0.3% on average).

Cluster 4. Layers in this cluster have (a) a larger parameter footprint (0.5–2.5 MB), (b) low-to-moderate FLOP/B ratio for parameters (25–64), and (c) moderate MAC intensity (5M–25M). The layers exhibit small activation footprints but rather high activation reuse. A large portion of layers in this category are standard convolution layers with deep input/output channels and input activation width/height, along with a large number of kernels. We find that these layers not only have large parameter footprint, but also include the majority of parameters (up to 64.3%) for their respective model. Cluster 4 layers average a PE utilization of 32%.

Cluster 5. Layers in this cluster have (a) a very small parameter footprint (1–100 kB), (b) a moderate FLOP/B ratio for parameters (49–600), and (c) low MAC intensity (0.5M–5M). The layers exhibit rather high activation footprints but have almost zero activation data reuse. Many of these layers are depthwise layers, which have only one channel (hence the lack of activation reuse), but have high parameter reuse because a small number of filters are convolved across a large input activation (which is the opposite of Cluster 4 layers) Cluster 5 layers achieve a low average PE utilization of 21%.

6.4.2. Hardware Design Principles

The five clusters that we identify in Section 6.4.1 serve as a starting point for determining the number of accelerators that we need to implement. As we study the distinguishing characteristics of each cluster, we find that some characteristics have a strong influence on the hardware design, while others do not necessitate significant changes to the hardware. We discuss two insights that drive our hardware design decisions.

First, we find that MAC intensity and parameter footprint/reuse drive a significant divergence in hardware design, as they impact a number of key accelerator design parameters (e.g., PE array size, on-chip buffer size, memory bandwidth considerations). Looking at our five clusters, we identify that (a) layers in Clusters 1 and 2 share a high MAC intensity, smaller parameter footprint, and moderate-to-high parameter reuse; while (b) layers in Clusters 3 and 4 share a lower MAC intensity, larger parameter footprint, and low parameter reuse. This means that we need *at least* two different accelerator designs: one that caters to the more compute-centric behavior of Clusters 1/2, and one that caters to the more data-centric behavior of Clusters 3/4. Given our resource-constrained environment, we look to see if layers in Cluster 5, which have a low MAC intensity (similar to Clusters 3 and 4) but a smaller parameter footprint (similar to Clusters 1 and 2), can benefit from one of these two approaches. We find that the low MAC intensity, along with the lower parameter reuse by many Cluster 5 layers, allow the layers to benefit from many of the non-compute-centric optimizations that benefit Clusters 3 and 4, so we consider them together.

Second, while there are many different types of accelerators that one can build for both compute-centric and data-centric layers, the reuse patterns of layers are a key distinguishing factor between these designs. The dataflow of a particular accelerator has a very strong impact on PE utilization and energy efficiency, as the dataflow dictates whether spatial and/or temporal reuse opportunities in layers are exploited. Prior work [224] analyzes the large dataflow design space, and discusses four types of data reuse that different dataflows can exploit: *spatial multicasting* (reading a parameter once, and spatially distribute it as an input to multiple PEs), *temporal multicasting* (replicating a parameter in a small local buffer, and delivering the parameter as multiple inputs at different times to the same PE), *spatial reduction* (accumulating activations from multiple PEs at the same time using multiple compute units), and *temporal reduction* (accumulating multiple activations generated at different times using a single accumulator/buffer). The chosen dataflow directly affects how the memory system and on-chip network of an accelerator should be designed. Thus, for both our compute-centric and data-centric clusters, we develop different dataflows for clusters with significantly different reuse patterns.

Both of our compute-centric clusters share high FLOP/B ratios for both parameters and activations, have a small parameter footprint, and a high activation footprint (relative to layers in other clusters). As a result, layers from both clusters benefit from a similar accelerator dataflow, which exposes reuse opportunities for both parameters and activations. Between the compute-centric optimizations and the shared dataflow affinity, we determine that we can use a single accelerator (*Pascal*) to efficiently execute layers from both Cluster 1 and Cluster 2.

Across our three data-centric clusters, we find that Clusters 4 and 5 share common behaviors related to data reuse: small footprints and low reuse for activation, with large footprints and moderate reuse for parameters. Thus, these layers benefit from a dataflow that exposes reuse opportunities for parameters and can share an accelerator (*Jacquard*). Cluster 3 layers exhibit different data reuse characteristics compared to the other two. Layers from Cluster 3 have extremely low parameters FLOP/byte ratio but very large footprint. We find that these layers benefit from a dataflow that provides temporal reduction opportunities for activations. As a result, we need a separate accelerator (*Pavlov*).

We employ a *template-based* design approach: while we design each accelerator based on cluster characteristics, we maintain the same generic tiled architecture as the baseline accelerator.

We do this to ease the integration of our hardware into a real system: from the perspective of compilers and programs, each of our accelerators appears to be just a different instance (with a different configuration) of the baseline accelerator. This is a *critical design decision*, as it allows us to deploy and run models using existing highly-optimized design/compile toolchains (e.g., Edge TPU compiler [115]) seamlessly on all of Mensa's accelerators, with the trade-off of limiting the degree of customization (and, thus, efficiency) our accelerator designs can achieve. While Mensa can work with accelerators that do not employ this tiled architecture, we make this design choice to avoid the need for more complex compilers and multiple libraries for programmers.

6.4.3. Pascal Accelerator Design

Dataflow. As we discuss in Section 6.4.2, Pascal caters to Cluster 1/2 layers, which have opportunities for both parameter and activation reuse. The layers have a small parameter footprint with high reuse, which can be captured by a very small on-chip buffer. As a result, exposing reuse opportunities for parameters will likely not provide much additional benefit. The large output activation footprint would require a relatively large buffer to effectively exploit reuse, which would consume significant area and static energy. We can avoid these inefficiencies by having the dataflow provide temporal reduction opportunities for the activations, which would significantly reduce the required buffer size while still effectively enabling high reuse (*first requirement*).

We consider a wide range of dataflow designs that can enable temporal reduction. One example is a dataflow where activations are spatially distributed across PEs, allowing the PEs to collaboratively calculate each output (by spatially multicasting parameters to each PE every cycle). We illustrate

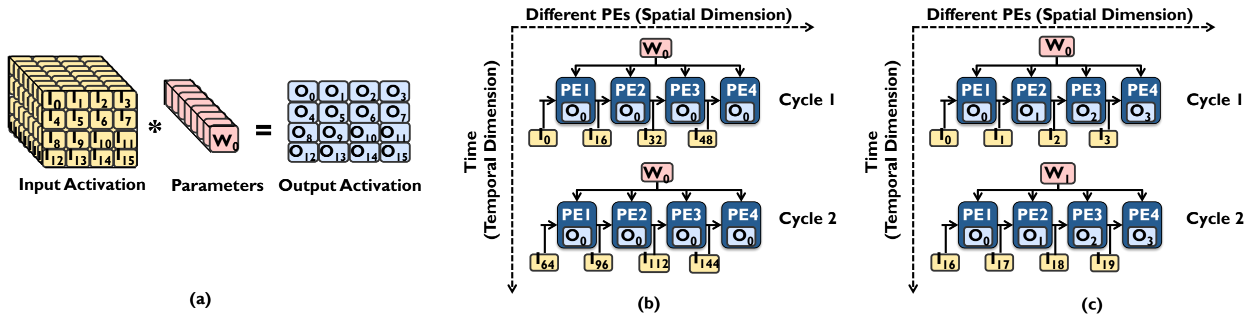


Figure 6.7. (a) Simplified pointwise layer, (b) an example dataflow that generates high network traffic by spatially reducing output activations, (c) our proposed dataflow.

how the components of a pointwise layer (Figure 6.7a) would map to such a dataflow in Figure 6.7b. The accelerator would accumulate partial sums from multiple PEs (four PEs per cycle in the figure), storing the results in the on-chip buffer. However, this approach actually worsens the impact of the large footprint of output activations, as each output activation is now split into multiple partial sums (one per PE), and each partial sum needs to traverse the on-chip network when the sums are gathered and reduced. For a representative pointwise layer with 512x512 output activations and an input channel depth (K) of 8, the partial sums alone generate 8 MB of on-chip traffic (K partial sums per output activation element). This worsens for a standard convolution layer, where there would be MxNxK partial sums per output activation element (where M and N are the input activation dimensions). Given the high traffic generated by partial sums, we instead need to use a dataflow with temporal reduction that *avoids* spatial reduction for output activations (*second requirement*).

We use our two design requirements to develop the dataflow for Pascal. Figure 6.7c shows a toy example of how Pascal's dataflow works for an accelerator with four PEs. We spatially distribute output activations, such that each PE calculates one output activation element. This enables temporal reduction for each output activation, and reduces the on-chip buffer size. We temporally replicate each parameter across PEs at each time step (i.e., each PE receives the same parameter), while spatially distributing input activations across PEs. This provides spatial reuse (multicast) for parameters across all PEs in each cycle. Our proposed dataflow generates no partial sum traffic, as the partial sums are instead *temporally reduced* in each PE's register file.

PE Array. Layers from Clusters 1 and 2 perform a large number of MAC operations, so we want to size Pascal's PE array to efficiently perform these in parallel. We analyze the inference latency using different array configurations, and empirically find that a 32x32 array strikes a balance between latency, PE utilization, and energy consumption. In particular, a smaller array increases the latency for Cluster 1 layers, while a larger array provides few benefits. With this array, Pascal achieves a 2 TFLOP/s peak throughput.

Memory System. To maximize energy efficiency and utilization, we co-design the proposed dataflow with the accelerator's memory system. The data reuse opportunities exposed by our proposed dataflow enable us to significantly reduce the on-chip buffer sizes, significantly reducing the energy and area costs of Pascal. As we mentioned above, by enabling temporal reduction (in output activations) with our dataflow, we can reduce the size of the on-chip buffer that holds the activations from 2 MB down to 256 kB. Thanks to the small parameter footprint of layers in Clusters 1 and 2, we can reduce the on-chip parameter buffer by 32x, down to 128 kB. Since Pascal falls under compute-intensive category, we assume that off-chip bandwidth available to PE array is same as the baseline accelerator (conventional LPDDR4 channel).

6.4.4. Pavlov Accelerator Design

Dataflow. We leverage three characteristics of Cluster 3 layers to shape the design of Pavlov: (1) parameters exhibit zero reuse but have a very large footprint, (2) activations exhibit high reuse but have a small footprint, and (3) the main operation is MVM. Based on these, we identify two design requirements for our dataflow.

First, our dataflow should expose activation reuse opportunities during MVM. One potential way to do this is using spatial reduction for output activations (i.e., all PEs collaboratively calculate one output element), with the input vector spatially distributed at each iteration. However, due to the large parameter footprint (e.g., W_h in LSTM cells), this would experience similar issues as we saw for spatial reduction for Clusters 1 and 2 (Section 6.4.3): there would be a large number of partial sums that need to be combined (as many as one per PE every cycle), resulting in a large amount of network traffic. As a result, we look for more efficient ways to expose activation reuse.

Second, we need to effectively utilize memory bandwidth at low cost. The on-chip buffer is ineffective for these layers, due to the lack of data reuse and the large parameter footprint. As a result, high PE utilization requires many off-chip accesses, and, thus, access to high bandwidth, such as the bandwidth available *inside* modern 3D-stacked memories. While we want to fully utilize this bandwidth, we cannot do so simply by issuing many outstanding memory requests at once from the accelerator, as this requires complex hardware that often exceeds the area and power budgets available in these memories (and in edge devices) [42, 77, 99]. However, if we can design our dataflow to issue *sequential* memory accesses to parameters, we can exploit this sequential pattern to use the bandwidth without complex hardware and at a much lower energy cost [77, 99].

Using these requirements, we design our Pavlov dataflow. Output activations are spatially distributed such that each PE is responsible for one output element. Every cycle, the dataflow replicates each input element across all PEs. We use spatial parameter distribution, with each PE receiving a different parameter element during a cycle. Figure 6.8a shows a toy example of MVM for the forget gate in a LSTM cell, while Figure 6.8b shows an example of how Pavlov's dataflow runs that MVM operation on an accelerator with 4 PEs. Each cycle, all of the PEs multiply one input vector element ($X_0(t)$ in this example) with a sub-row of W_x (or W_h), storing the partial sum in the PE register file. In the next cycle, the next input element ($X_1(t)$) is replicated across all PEs, the next sub-row of parameters is spatially distributed, and the multiply is performed. The product is added to the partial sum sitting in the PE register file. After we multiply all rows of the parameter matrix with input vector elements, we obtain the final output activation in each PE. This dataflow allows us to (1) provide temporal reduction for output activations using the PE register file, and (2) access parameters sequentially.

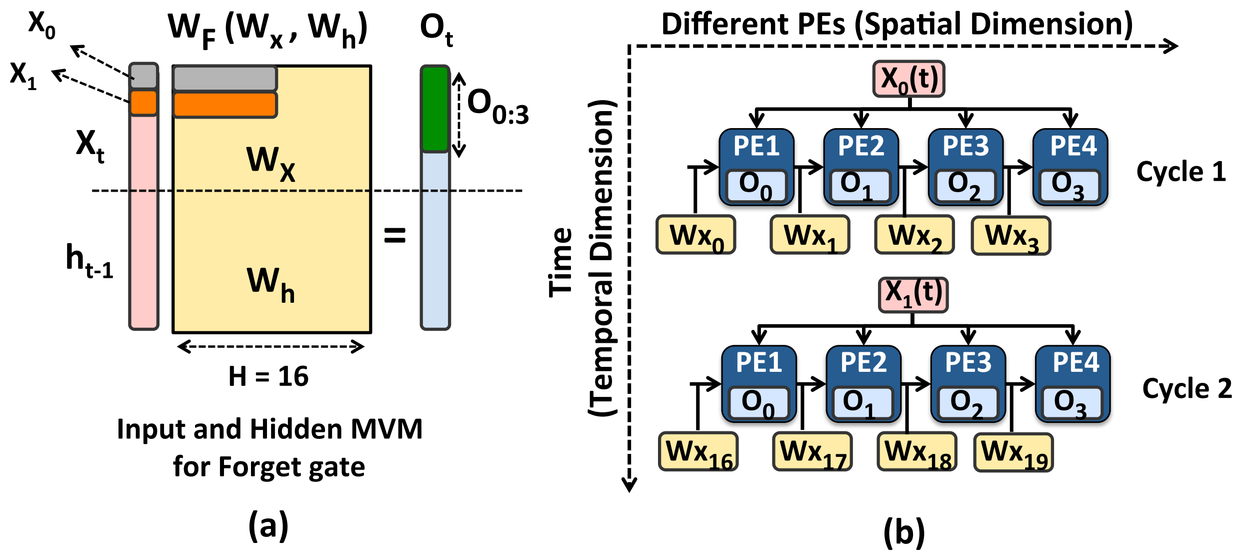


Figure 6.8. (a) Input/hidden MVMs for a forget gate in a LSTM cell, (b) our proposed dataflow for an LSTM cell.

While the above dataflow is efficient for a single MVM operation, it does not yet account for inter- and intra-cell dependencies in LSTM layers (which comprise the majority of Cluster 3 layers). These dependencies provide further opportunities for reuse that, if not exploited, can be costly. Without dependency-based reuse, if we have an LSTM where the input sequence length is 100, and the dimensions of W_x and W_h are both 1000x1000, a single layer would generate 500 MB of off-chip parameter traffic.

Even though W_x and W_h elements are not reused within an LSTM cell, they are reused across LSTM cells within a layer (Figure 6.9a). For example, W_{hf} is reused across all cells for the forget gate. Unfortunately, we cannot simply cache all of W_{hf} on-chip, given its size. Instead, we exploit two observations: (1) there are no dependencies between the input MVMs across cells, and (2) we know the input sequence (e.g., x_{t-1}, x_t, x_{t+1}) ahead of time. As a result, we can decouple the computation of input and hidden MVM in each gate, and calculate all input MVMs for all cells ahead of time by fetching W_{hf} elements only once from memory. Then, we later perform the hidden MVM computations while still respecting inter-cell dependencies between them.

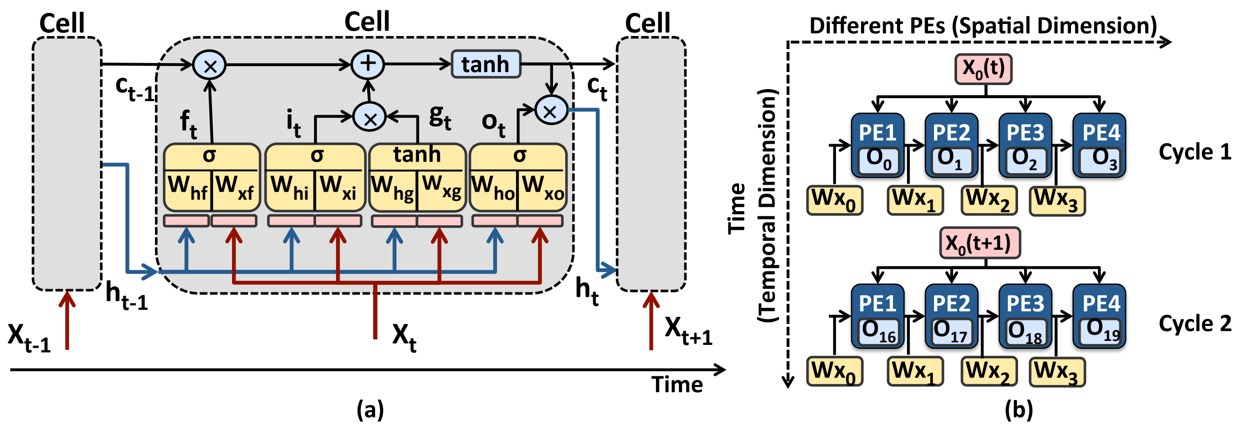


Figure 6.9. (a) LSTM layer internal structure, (b) our LSTM cell dataflow optimized to exploit dependencies.

We modify our dataflow, as shown in Figure 6.9b, to efficiently reuse W_x and W_h. Now, when we fetch and spatially distribute a sub-row of W_x across PEs, we keep those parameters in the PE register file. Each iteration, every PE multiplies one element of x_t ($x_0(t)$ in Figure 6.9b) and one element of the W_x sub-row. Now, instead of going to the next sub-row, we fetch the next input element ($x_0(t+1)$ in Figure 6.9b) and perform the MVM for another LSTM cell. Note that instead of accumulating only a single partial sum in each PE's register file, we accumulate K partial sums, one for each LSTM cell being computed concurrently. With this dataflow, we eliminate the need to fetch W_x more than once from DRAM.

PE Array. Because Cluster 3 layers have a low MAC intensity, and mainly perform MVM, we design a much smaller PE array for Pavlov than that in the baseline accelerator. We again analyze the inference latency across a range of PE array sizes, and find that an 8x8 array (128 GFLOP/s peak throughput) balances latency, utilization, and energy.

Memory System. We design Pavlov's memory system to be aware of (1) its dataflow and (2) Cluster 3 layer characteristics. Since Pavlov is data-centric and its sequential memory accesses exploit high memory bandwidth, we place it in the logic layer of a 3D-stacked memory, where the bandwidth is much higher than the external bandwidth to the CPU. Given that parameters and activations from these layers exhibit different characteristics, we customize separate on-chip buffers for each data type. For parameters, we use only one level of memory hierarchy (512 B private buffer per PE) and stream parameters directly from DRAM, as (1) their minimal reuse and large footprint does not warrant a shared on-chip buffer, (2) our proposed dataflow allows us to fully utilize 3D-stacked memory bandwidth with simple hardware at low cost (sequential accesses are much less costly than random accesses [77, 99]). For activations, thanks to the small activation footprint of layers in Cluster 3, we use a 128 kB buffer.

6.4.5. Jacquard Accelerator Design

Dataflow. We leverage three characteristics of Cluster 4/5 layers to shape the design of Jacquard: (1) activations exhibit very low reuse; (2) parameters exhibit lower reuse than Cluster 1/2 layers, but exhibit much higher reuse than activations; and (3) the footprint of activations is low for all layers, while the parameter footprint is high for many (but not all) layers. Based on these, we need to design a dataflow that exposes reuse opportunities for parameters, which in turn allows us to shrink the on-chip parameter buffer.

There are several variations of dataflows that can enable reuse opportunities for parameters. For example, we can replicate the same parameter across different PEs, and temporally reuse parameter across different iterations. For each parameter, this results in a reuse factor (i.e., the number of cycles a parameter is reused) of WxH/N, where N is the number of PEs and WxH is the dimension of the output activations. Once we traverse all WxH output activations, we need to broadcast a new parameter element across the PEs and repeat the process. If the output activation dimension is small (which is the case for Cluster 4/5 layers), the reuse factor for each parameter becomes too small, resulting in frequent parameter broadcasting. This is particularly problematic for Cluster 4 layers, which often have a large number of filters (i.e., a large parameter footprint). The on-chip traffic generated by frequent parameter broadcasting prevents us from efficiently overlapping the retrieval of parameters from DRAM with PE computation, leading to PE underutilization.

To avoid this in Jacquard, we instead spatially distribute parameters across PEs. All PEs collaboratively calculate one output activation, while input activations are spatially distributed among PEs. The parameters stay in the PE register file across different cycles (temporal reuse) until we traverse the all of the output activations. This increases the reuse factor to WxH and reduces on-chip traffic (by eliminating the parameter broadcasts). With this larger reuse factor, our dataflow can now effectively hide the DRAM access latency with PE computation time.

PE Array. While Cluster 4/5 layers have a low MAC intensity, they perform more MAC operations on average than Cluster 3 layers. Our analysis of PE array sizes shows that equipping Jacquard with an array smaller than 16x16 increases the latency, so we select 16x16 for the array size. This allows Jacquard to achieve a peak throughput of 512 GFLOP/s.

Memory System. Similar to Pavlov, we co-design Jacquard's memory system with its dataflow, and decide to place Jacquard inside the logic layer of 3D-stacked memory. This provides high bandwidth for the large parameter footprints of Cluster 4 layers. We use separate shared buffers for each data type. Given the small activation footprints, we use a small 128 kB buffer for them (a 16X reduction compared to the baseline accelerator). Our proposed dataflow for Jacquard enables temporal reuse for parameters across different iterations. Thanks to this temporal reuse and layer characteristics, we significantly reduce the parameter buffer size by 32X (down to 128 kB) compared to the baseline. Our much smaller buffers significantly reduce the area and energy consumption of Jacquard over the baseline accelerator.

6.5. Experimental Methodology

6.5.1. Models & Simulation

We analyze 24 Google edge NN models, which are used as benchmarks for the Edge TPU. These models are used in several Google mobile applications/products, such as image classification, object detection, semantic segmentation, automatic speech recognition, and image captioning. The models are specifically developed for edge devices using TensorFlow Lite [127]. All models are fully 8-bit quantized (i.e., 32-bit floating point tensor parameters are converted into 8-bit fixed-point numbers) using a quantization-aware training approach [129]. The models are then compiled using the Edge TPU compiler [115], which generates custom operations that execute on the Edge TPU.

6.5.2. Energy Analysis

We build our energy model based on prior works [42, 43, 99], which sums up the total energy consumed by the accelerator, DRAM, off-chip and on-chip interconnects, and all on-chip buffers. We use CACTI-P 6.5 [270] with a 22 nm process technology to estimate on-chip buffer energy. We assume that each 8-bit MAC unit consumes 0.2 pJ/bit. We model the DRAM energy as the energy consumed per bit for LPDDR3, using estimates and models from prior works [42, 99].

6.5.3. Performance Analysis

For performance analysis, we use an in-house simulator that models the Edge TPU architecture. The simulator models major components of the accelerator including the PE array, memory system, on-chip network, and dataflow. We heavily modified the simulator to implement our three proposed accelerators and the software runtime of Mensa. We develop an analytical cost model to determine the performance of each of our proposed dataflows, and integrate the dataflow performance numbers into our simulator's performance model. We use CACTI-P 6.5 [270] to adjust buffer access time latency for each of the on-chip buffers in our proposed accelerators. Like prior work [42, 43, 77, 99], we use the memory bandwidth available inside a 3D-stacked memory chip as the memory bandwidth available to the data-centric accelerators (which reside in the logic layer of the memory). We model a 3D-stacked memory based on the High-Bandwidth Memory (HBM) standard [186], and use the bandwidth available in the logic layer of HBM (256 GB/s), which is 8x more than the bandwidth available (32 GB/s) to the compute-centric accelerator and the baseline edge ML accelerator. To demonstrate that high bandwidth alone is not enough to solve the efficiency issues of the baseline accelerator, we add a hypothetical configuration to our evaluation (*Base+HB*), where the baseline accelerator has access to the same 8x internal bandwidth as logic in the logic layer of HBM.

6.6. Evaluation

We evaluate energy and performance for four configurations: (1) *Baseline*, the Edge TPU; (2) *Base+HB*, a hypothetical version of Baseline with 8x bandwidth (256 GB/s); (3) *EyerissV2*, a state-of-the-art edge accelerator [57] that uses reconfigurable interconnects to address CNN model heterogeneity; and (4) our implementation of *Mensa* with all three proposed accelerators (Pascal, Pavlov, Jacquard). To improve figure clarity, we show individual model results for only a few representative models of each model type. Our average results are reported across *all* 24 Google edge models.

6.6.1. Inference Energy Analysis

Figure 6.10 (left) shows the total inference energy across different NN models by our four configurations.

We make three observations from the figure.

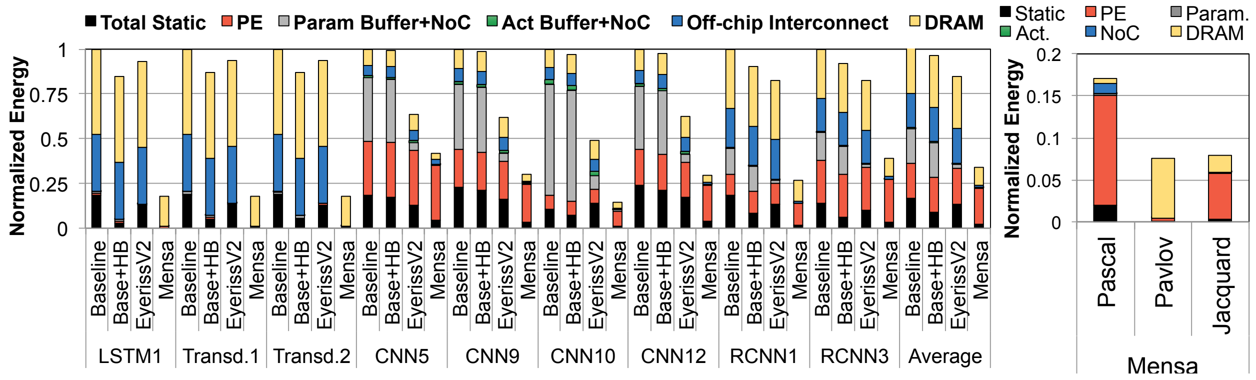


Figure 6.10. Inference energy across different models (left) and energy breakdown across our three proposed accelerators (right), normalized to Baseline.

First, providing high bandwidth to Baseline results in only a small reduction (7.5% on average) of inference energy. This is because Base+HB still incurs the high energy costs of (1) on-chip buffers that are overprovisioned for many layers, and (2) off-chip traffic to DRAM. Base+HB benefits LSTMs and Transducers the most (14.2% energy reduction), as the higher bandwidth significantly reduces the inference latency of these models, which in turn lowers static energy.

Second, EyerissV2 suffers from significant energy inefficiency for LSTMs and Transducers. While EyerissV2 lowers static energy compared to Baseline, due to its use of a much smaller PE array (384 vs. 4096) and on-chip buffers (192 kB vs. 4 MB), it still incurs the high energy costs of off-chip parameter traffic to DRAM. Averaged across all LSTM and Transducer models, EyerissV2 reduces energy by only 6.4% over Baseline. For CNN models, EyerissV2 reduces inference energy by 36.2% over Baseline, as its smaller on-chip buffer significantly reduces dynamic energy consumption.

Third, Mensa significantly reduces inference energy across all models. The reduction primarily comes from three sources. (1) Mensa lowers the energy spent on on-chip and off-chip parameter traffic by 15.3x, by scheduling layers on the accelerators with the most appropriate dataflow for each layer. LSTMs and Transducers benefit the most, as their inference energy in both Base+HB and EyerissV2 is dominated by off-chip parameter traffic, which Pavlov and Jacquard drastically cut due to being placed inside memory. (2) Mensa reduces the dynamic energy of the on-chip buffer and network (NoC) by 49.8x and 6.2x over Base+HB and EyerissV2, by avoiding overprovisioning and catering to specialized dataflows. This is most beneficial for CNN and RCNN models. (3) Mensa reduces static energy by 3.6x and 5.6x over Base+HB and EyerissV2, thanks to using significantly smaller PE arrays that avoid underutilization, significantly smaller on-chip buffers, and dataflows that reduce inference latency.

EyerissV2 falls significantly short (50.6%) of Mensa's energy efficiency for three reasons. First, while EyerissV2's flexible NoC can provide a high data rate to the PE array, its fixed dataflow cannot efficiently expose reuse opportunities across different layers (e.g., Cluster 4 and 5 layers that have very large parameter footprints and low data reuse). Second, EyerissV2 has much higher static energy consumption, as its inference latency is significantly larger for many compute-intensive

CNN layers (as its PE array is much smaller than the array in Pascal). Third, some CNN layers have a large parameter footprint and very low data reuse, which generates a large amount of off-chip parameter traffic in EyerissV2. Overall, Mensa reduces total inference energy by 66.0%/50.6%, and improves energy efficiency (TFLOP/J) by 3.0x/2.4x, compared to Baseline/EyerissV2.

Figure 6.10 (right) shows the breakdown of energy usage across our three Mensa accelerators. Pascal consumes the most energy of the three, with its consumption dominated by the PE array (since the layers that run on Pascal perform a large number of MAC operations). Pavlov's energy usage is dominated by DRAM accesses, as its layers have large footprints and no data reuse. For Jacquard, the majority of energy is used by a combination of DRAM accesses and the PE array, but the usage is lower than Pavlov DRAM accesses or the Pascal PE array due to the inherent layer properties (smaller footprints, lower MAC intensity).

6.6.2. Performance Analysis

Figure 6.11 shows the utilization (bars, left axis) and throughput (lines, right axis) for our four configurations. Mensa's utilization is calculated by computing the average utilization across its three accelerators (Pascal, Pavlov and Jacquard). We make three observations from the figure.

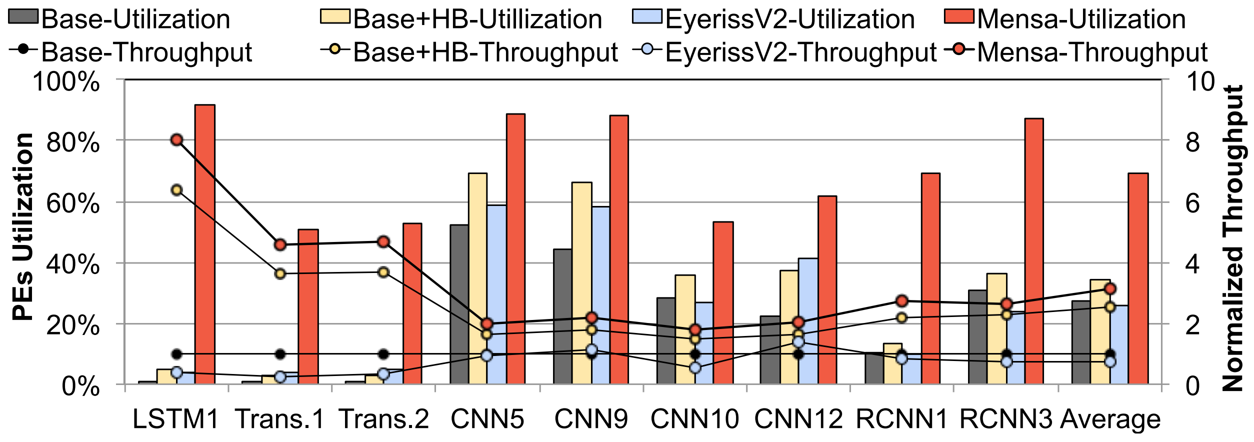


Figure 6.11. Accelerator utilization and throughput across different models, normalized to Baseline.

First, Baseline suffers from low PE array utilization (on average 27.3%). The higher bandwidth in Base+HB pushes average utilization up to 34.0%, and improves throughput by 2.5x. The largest improvements are for LSTMs and Transducers (4.5x on average vs. 1.3x for CNNs), thanks to their low FLOP/B ratio and large footprints. In contrast, some CNN models (e.g., CNN10) see only modest improvements (11.7%) with Base+HB, as their layers have high reuse and small footprints. Overall, Base+HB still has very low utilization, as many layers (those from Clusters 3, 4, and 5) do not need the large number of PEs in the accelerator.

Second, EyerissV2 *reduces* performance significantly over Baseline for several models. EyerissV2's flexible interconnect and much smaller PE array do allow it to achieve slightly higher PE utilization than Baseline for layers that have very low data reuse. However, this higher utilization is offset by significantly higher inference latencies. For compute-intensive layers in Clusters 1 and 2, the smaller PE array size hurts layer throughput. For data-centric layers in Clusters 4 and 5, EyerissV2 cannot customize its dataflow to expose reuse opportunities, and thus is hurt by the high off-chip traffic. Overall, we find that EyerissV2's overall throughput is actually *lower* than Baseline for most of our models.

Third, Mensa provides a significant increase in both average utilization (2.5x/2.0x/2.6x) and throughput (3.1x/1.3x/4.3x) over Baseline/Base+HB/EyerissV2. The large utilization improvements are a result of (1) properly-provisioned PE arrays for each layer, (2) customized dataflows that exploit reuse and opportunities for parallelization, and (3) the movement of large-footprint layer computation into memory (eliminating off-chip traffic for their DRAM requests). We note that Mensa's throughput improvements over Base+HB are smaller than its utilization improvements, because Base+HB is reasonably effective at reducing the inference latency of layers with poor reuse and large footprints (albeit with poor energy efficiency and underutilization). Mensa benefits all NN model types, but the largest improvements are for LSTMs and Transducers, with average utilization/throughput improvements of 82.0x/5.7x over Baseline. The improvement is lower for CNNs and RCNNs (2.23x/1.8x over Baseline), because they make more use of Baseline's large PE arrays, and have smaller footprints that lessen the impact of off-chip DRAM accesses. For a few CNNs (CNN10–CNN13), their utilization is somewhat lower than desired (44.7%) due to their use of a large number of depthwise layers (part of Cluster 5) that have significantly lower data reuse than other Cluster 4/5 layers. While these layers run less optimally with Jacquard's dataflow due to the different reuse behavior, Mensa still improves their utilization by 65.2% over Baseline because

Jacquard's specialization still helps depthwise layers.

Figure 6.12 shows the overall inference latency, and where the inference latency in Mensa is spent (across Pascal, Pavlov, and Jacquard). We find that Mensa outperforms Baseline and Base+HB on average by 1.96x and 1.17x. LSTMs and Transducers see a significant latency reduction with Mensa (5.4x/1.26x vs. Baseline/Base+HB) because most of their layers run on Pavlov and benefit from an optimized dataflow and processing-in-memory (which provides not only higher bandwidth, but also lower latency for DRAM accesses). CNNs and RCNNs benefit from the heterogeneity of our accelerators, making use of all three of them to reduce latency by 1.64x/1.16x over Baseline/Base+HB.

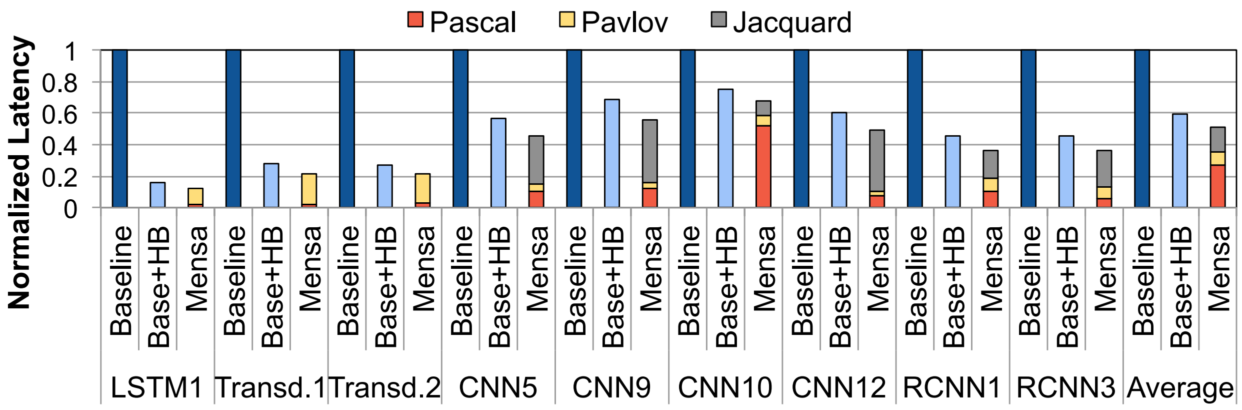


Figure 6.12. Inference latency across different models, normalized to Baseline.

6.7. Summary

We conduct the first bottleneck analysis of the Google Edge TPU, a state-of-the-art ML inference accelerator, as it executes 24 state-of-the-art Google edge NN models. Our analysis reveals that the Edge TPU's monolithic design leads to significant underutilization and poor energy efficiency for our edge NN models, which exhibit significant layer variation. We propose a new framework called Mensa, consisting of multiple small heterogeneous accelerators, each specialized to specific layer characteristics. Using our discovery that layers group into a small number of clusters, we create a Mensa design for the Google edge NN models consisting of three accelerators. Compared to the Edge TPU and to a state-of-the-art reconfigurable ML accelerator, our design improves energy efficiency by 3.0x/2.4x and throughput by 3.1x/4.3x for our edge NN models. We hope that Mensa can enable the design and adoption of future heterogeneous accelerators that support yet-to-be-developed NN model types.

Chapter 7

Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design

Data analytics involves the processing and analysis of large amounts of data to identify a wide range of patterns and trends. It has become popular due to the exponential growth in data generated annually [6]. Many application domains need to perform *real-time data analysis* (i.e., perform analytics on the most recent version of the data) [4,106,337]. A major example is Internet-of-Things (IoT) applications, which can ingest a large volume of data from various sensors, and typically have some form of learning model (e.g., SQL analytics, machine learning, graph analytics) that is applied to the ingested data to make in-the-field decisions (e.g., navigation for self-driving cars, patient monitoring and response in healthcare) [7,29,30,33]. For such applications, the need to analyze data in real time is critical, as the data's value diminishes significantly over time.

Analytics often make use of *database management systems* (DBMSs) for the underlying data. An analytical workload includes a series of queries that perform analytics on a particular set of fields across *many* database records. An analytics DBMS optimizes its data storage to cater to these queries, which can be complex and long, and perform predominantly read-only operations. For example, for SQL analytics, database tables are stored in column-major format (as each field corresponds to a column), and an analytical workload scans a select number of these columns across many database rows [176, 348].

An analytics DBMS differs significantly from the traditional transactional DBMS, due to the different nature of their respective workloads. A transactional workload typically includes queries (i.e., transactions) that perform one or more lookups or changes to individual records, potentially accessing multiple fields in each record. A transactional DBMS optimizes its data storage to transactions, which are typically short-lived and exhibit bursty write behavior. For example, for SQL transactions, database tables are stored in row-major format [193, 349], as each row corresponds to a single record.

Many applications require a combination of analytics and transactions. For example, a self-driving car uses transactions to record each periodic sample of data from all sensors, while running analytics across specific sensor data over time to analyze the car's surroundings and make real-time navigation decisions. With separate DBMSs, new transactions are propagated to the analytics DBMS using a costly process that is performed only on the order of hours or days [205, 255], causing analytics to often be performed on *stale* data [4, 30, 205, 255]. As this is intolerable for real-time analysis, *hybrid transactional/analytical processing* (HTAP) DBMSs [4, 106, 337] have emerged, providing a single (typically in-memory) DBMS for transactions and analytics [4, 30, 231, 255, 288].

To support both types of workloads with high throughput and minimal energy, an ideal HTAP system should have three properties [255]. First, it should ensure that transactional and analytical workloads benefit from their own workload-specific optimizations (e.g., algorithms, data structures). Second, it should guarantee data freshness (i.e., access to the most recent version of data) for analytical workloads while ensuring that transactional and analytical workloads have a consistent view of data across the system. Third, it should ensure that the latency and throughput of transactional and analytical workloads are the same as if they were run in isolation.

Our goal is to develop an HTAP system that achieves all three of the desired HTAP properties, with new architectural techniques and awareness of PIM capability.

7.1. Summary of Key Insights and Observations

Meeting all three ideal HTAP properties at once is very challenging, as transactional and analytical workloads have different underlying algorithms and access patterns, and optimizing for one property can often require a trade-off in another property. We extensively study state-of-the-art in-memory HTAP systems, and find that no system exists that can meet all three properties. We observe two key problems that prevent state-of-the-art HTAP systems from simultaneously achieving the three desired properties.

First, these systems experience a drastic reduction in transactional throughput (up to 74.6%) and analytical throughput (up to 49.8%) compared to when we run each in isolation. This is because the mechanisms used to provide data freshness and consistency induce a significant amount of *data movement* between the CPU cores and main memory.

Second, HTAP systems often fail to provide effective performance isolation. These systems suffer from severe performance interference (up to 31.3% reduction in transactional throughput) because of the high resource contention between transactional workloads and analytical workloads.

Unfortunately, as our evaluation shows, simply providing more memory bandwidth to alleviate data movement costs and memory contention cannot solve these challenges; we need careful architectural design and hardware–software cooperative approaches.

To solve the challenges faced by existing HTAP systems, we propose a novel hardware/software cooperative design for in-memory HTAP databases called Polynesia. The key idea of Polynesia is to partition the computing resources in a system into two types of isolated, specialized processing *islands* (*transactional islands* and *analytical islands*). By isolating transactional islands from analytical islands, we are able to (1) apply workload-specific optimizations to each island; (2) avoid high resource contention; and (3) design new and efficient mechanisms that propagate data updates from transactional islands to analytical islands, where we can provide data freshness and consistency without incurring high data movement costs. As a result, to our knowledge, Polynesia is the first work to achieve all three desired properties of an HTAP system.

Figure 7.1 shows a high-level overview of Polynesia. Each island in Polynesia consists of (1) a replica of data for a specific workload, (2) an optimized execution engine (i.e., the software that executes queries), and (3) a set of hardware resources (e.g., computation units, memory) that cater to the execution engine and its memory access patterns. A transactional island is designed to sustain the bursts of writes performed as new data records are added to the database. The island's execution engine is similar to conventional transactional engines [193, 385]. Since the engine performs many small writes that benefit from caching, the hardware resources in a transactional island include conventional multicore CPUs with multi-level caches and access to a shared main memory. An analytical island is designed to provide very high read throughput for the analytics algorithms. In conventional systems, analytical workloads generate a large amount of data movement between the CPU and main memory, which incurs a high performance and energy cost [202] and becomes a bottleneck for the workloads. To alleviate this, we take advantage of a customized version of *processing-in-memory* (PIM), where we design new logic that can be placed in or near a memory chip to provide the logic with low-latency access to the internal memory chip throughput (which is much larger than the typical memory throughput available to a CPU [174, 186, 233]).

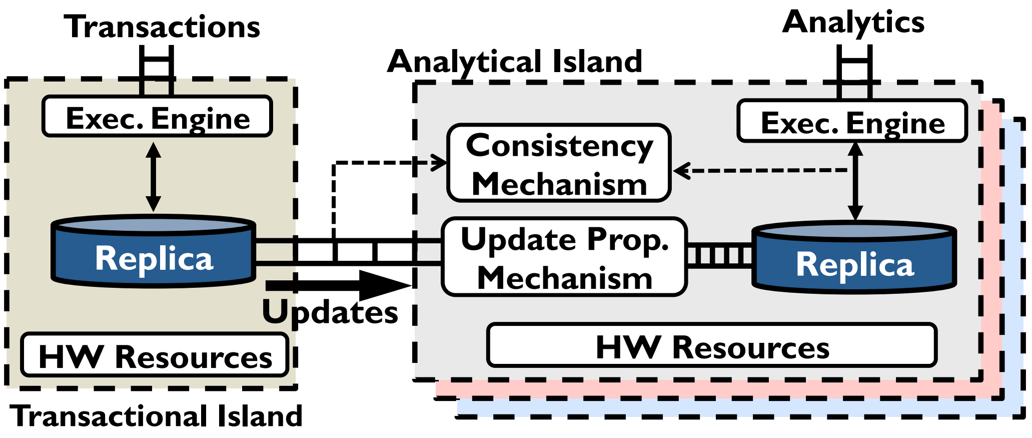


Figure 7.1. High-level architecture of Polynesia.

To solve the challenges of existing HTAP systems, we co-design new algorithms and efficient hardware support for the three key components of an analytical island: (1) an update propagation mechanism, which gathers updates from the transactional islands, converts updates to the analytical data format, and applies the updates to the analytical island's replica; (2) a consistency mechanism, which ensures that analytics queries observe a consistent view of data (i.e., an analytical workload will not see a newer update unless all prior updates are also visible); and (3) an analytical execution engine, which performs the query and handles query-related logistics. We design our update propagation mechanism, to take advantage of PIM, decoupling propagation into two parts (update shipping and update application), developing new efficient algorithms for each part, and designing custom hardware accelerators for each part that sit inside the logic layer of 3D-stacked memory. We design our consistency mechanism (Section 6) to use a new algorithm that runs on a custom hardware accelerator for efficient replica versioning in the logic layer. Our analytical execution engine (Section 7) consists of simple PIM cores to execute each query, a custom data placement algorithm that optimizes how columns of data are placed in a 3D-stacked memory cube to balance parallelization opportunities with the need to replicate metadata, and a new scheduler that schedules workload tasks on PIM cores based on a combination of work balancing and data locality.

We compare Polynesia to three state-of-the-art HTAP systems. Polynesia outperforms all three existing systems. Across a series of synthetic workloads that allow us to precisely study the benefits of each component of Polynesia, Polynesia achieves a higher transactional (2.20X/1.15X/1.94X; mean of 1.70X) and analytical (3.78X/5.04X/2.76X; mean of 3.74X) throughput. Polynesia also achieves a higher transactional (2.31X/1.19X/1.78X; mean of 1.76X) and analytical (3.41X/4.85X/2.20X; mean of 3.48X) over the existing systems for TPC-C [2] and TPC-H [3] workloads. Polynesia consumes less energy than all three as well, 48% lower than the prior lowest-energy system. Overall, we conclude that Polynesia efficiently provides high-throughput, energy-efficient real-time analysis, and that it meets all three desired HTAP properties.

7.2. Motivation

There are two major types of HTAP systems: (1) single-instance design systems and (2) multiple-instance design systems. In this section, we study both types, and analyze why neither type can meet all of the desired properties of an HTAP system.

7.2.1. Single-Instance Design

One way to design an HTAP system is to maintain a single instance of the data, which both analytics and transactions work on, to ensure that analytical queries access the most recent version of data. Several HTAP proposals from academia and industry are based on this approach [25, 30, 92, 136, 205, 314]. While single-instance design enables high data freshness, we find that it suffers from three major challenges:

(1) High Cost of Consistency and Synchronization. Since analytical and transactional workloads work on the same instance of data concurrently, single-instance-based systems need to ensure that the data is consistent and synchronized. One approach to consistency is to let both transactions and analytics work on the same copy of data, and use locking protocols [87] to maintain consistency across the system. However, locking has two major drawbacks. First, it significantly reduces the update throughput of transactions [60, 279] and degrades data freshness, as it blocks transactions from updating objects that are being read by long-running analytics queries. Second, it can frequently blocks analytics, as with the high update rate of transactions, a transactional workload can often lock out the analytics workload, leading to a significant drop in throughput. To avoid these drawbacks,To avoid the throughput bottlenecks incurred by locking protocols [87], single-instance-based HTAP systems resort to either snapshotting [25, 205, 266, 331] or multi-version concurrency control (MVCC) [30, 279]. Unfortunately, we find that snapshotting and MVCCboth solutions have significant drawbacks of their own.

Snapshotting: Several HTAP systems (e.g., [25, 205, 266]) use a variation of multiversion synchronization, called snapshotting, to provide consistency via snapshot isolation [36,47]. Snapshot Isolation guarantees that all reads in a transaction see a consistent snapshot of the database state, which is the last committed state before the transaction started. These systems explicitly create snapshots from the most recent version of operational data, and let the analytics run on the snapshot while transactions continue updating the data.

We analyze the effect of state-of-the-art snapshotting [25, 368] on throughput, for an HTAP system with two transactional and two analytical threads (each runs on a separate CPU). Figure 7.2

(right) shows the transaction throughput with snapshotting, normalized to a zero-cost snapshot mechanism, for three different rates of analytical queries. We make two observations from the figure. First, at 128 analytical queries, snapshotting reduces throughput by 43.4%. Second, the throughput drops as more analytical queries are being performed, with a drop of 74.6% for 512 analytical queries. We find that the majority of this throughput reduction occurs because `memcpy` is used to create each snapshot, which introduces significant interference among the workloads and generates a large amount of data movement between the CPU and main memory. The resulting high contention for shared hardware resources (e.g., off-chip channel, memory system) directly hurts the throughput.

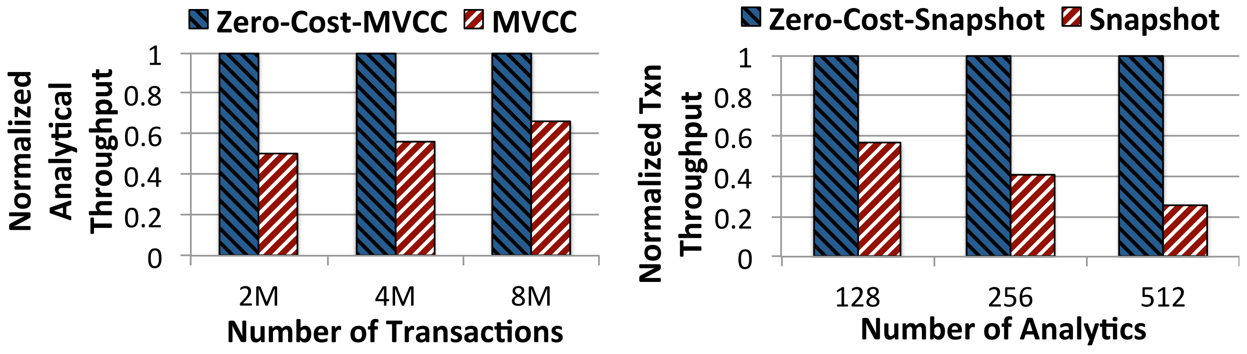


Figure 7.2. Effect of MVCC on analytical throughput (left) and snapshotting on transactional throughput (right).

MVCC: While both MVCC and snapshotting provide snapshot isolation, MVCC avoids making full copies of data for its snapshot. In MVCC, instead of replacing the old data on an update, the system keeps several versions of the data in each entry. The versions are chained together with pointers, and contain timestamps. Now, instead of reading a separate snapshot, an analytics query can simply use the timestamp to read the correct version of the data, while a transaction can add another entry to the end of the chain without interrupting. As a result, updates never block reads, which is the main reason that MVCC has been adopted by many transactional DBMSs (e.g., [92, 227, 230]).

However, MVCC is not a good fit for mixed analytical and transactional workloads in HTAP. We study the effect of MVCC on system throughput, using the same hardware configuration that we used for snapshotting. Figure 7.2 (left) shows the analytical throughput of MVCC, normalized to a zero-cost version of MVCC, for a varying transactional query count. We observe that the analytical throughput significantly decreases (by 42.4%) compared to zero-cost MVCC. We find that long version chains are the root cause of the throughput reduction. Each chain is organized as a linked list, and the chains grow long as many transactional queries take place. Upon accessing a data tuple, the analytics query traverses a lengthy version chain, checking each chain entry's timestamp to locate the most recent version that is visible to the query. As analytic queries touch a large number of tuples, this generates a very large number of random memory accesses, leading to the significant throughput drop.

(2) Limited Workload-Specific Optimization. A single-instance design severely limits workload-specific optimizations, as it is very challenging to enable optimizations for both types of workloads on a single replica of data.as the instance cannot have different optimizations for each workload. Let us examine the data layout in relational databases as an example. Relational transactional engines use a row-wise or N-ary storage model (NSM) for data layout, as it provides low latency and high throughput for update-intensive queries [193]. Relational analytics engines, on the other hand, employ a column-wise or Decomposition Storage Model (DSM) to store data, as it provides better support for columnar accesses, compression, and vectorized execution.

It is inherently impossible for a single-instance-based system to implement both formats simultaneously, and many such systems simply choose one of the layouts [25, 205, 314]. A few HTAP systems attempt to provide a hybrid data layout (e.g., [30]) or multiple data layouts in a single replica (e.g., [227]). However, these systems need to periodically convert data between different data formats, which leads to significant overhead and compromises data freshness [288].

(3) Limited Performance Isolation. It is critical to ensure that running analytics queries alongside transactions does not violate strict transactional latency and throughput service-level agreements. Unfortunately, running both on the same instance of data, and sharing hardware resources, leads to severe performance interference. We evaluate the effect of performance interference using the same system configuration that we used for snapshotting and MVCC. Each transactional thread executes 2M queries, and each analytical thread runs 1024 analytical queries. We assume that there is no cost for consistency and synchronization. Compared to running transactional queries in isolation, the transactional throughput drops by 31.3% when the queries run alongside analytics. This is because analytics are very data-intensive and generate a large amount of data movement, which leads to significant contention for shared resources (e.g., memory system, off-chip bandwidth). Note that the problem worsens with realistic consistency mechanisms, as they also generate a large amount of data movement.

7.2.2. Multiple-Instance Design

The other approach to design an HTAP system is to maintain multiple instances of the data using replication techniques, and dedicate and optimize each instance to a specific workload (e.g., [107, 215, 227, 230, 255, 266, 288]). Unfortunately, multiple-instance design systems suffer from a number ofseveral challenges:

Data Freshness. One of the major challenges in multiple-instance-based approach is to keep analytical replicas up-to-date even when the transaction update rate is high, without compromising performance isolation [106, 255]. To maintain data freshness, the system needs to (1) gather updates from transactions and ship them to analytical replicas, and (2) perform the necessary format conversion and apply the updates.

Gathering and Shipping Updates: Given the high update rate of transactions, the frequency of the gathering and shipping process has a direct effect on data freshness. During the update shipping process, the system needs to (1) gather updates from different transactional threads, (2) scan them to identify the target location corresponding to each update, and (3) transfer each update to the corresponding location. We study the effect of update shipping on transactional throughput for a multiple-instance-based HTAP system (see Section 7.7). Our system has two transactional and two analytical threads (each running on a CPU core). Figure 7.3 shows the transactional throughput for three configurations: (1) a baseline with zero cost for update shipping and update application, (2) a system that performs only update shipping, and (3) a system that performs both update shipping and update application (labeled as *Update-Propagation*). We observe from the figure that the transactional throughput of update shipping reduces by 14.8% compared to zero-cost update shipping and application. Our analysis shows that when the transactional queries are more update-intensive, the overhead becomes significantly higher. For update intensities of 80% and

100%, the throughput drops further (by 19.9% and 21.2%, respectively). We find that the reduction in throughputthis reduction is mostly because the update shipping process generates a large amount of data movement and takes several CPU cycles. Figure 7.4 shows the breakdown of execution time during update propagation process. We find that update shipping accounts for on average 15.4% of the total execution time.

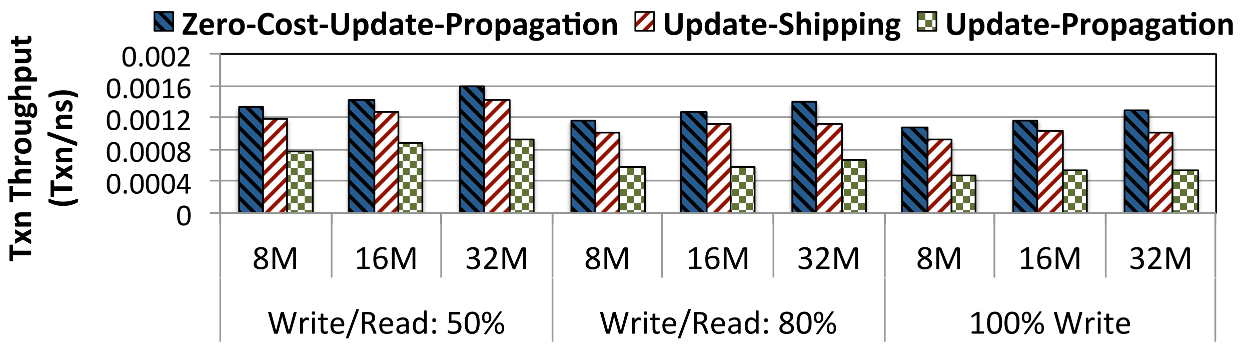


Figure 7.3. Transactional throughput across different number of transactions and different write intensities.

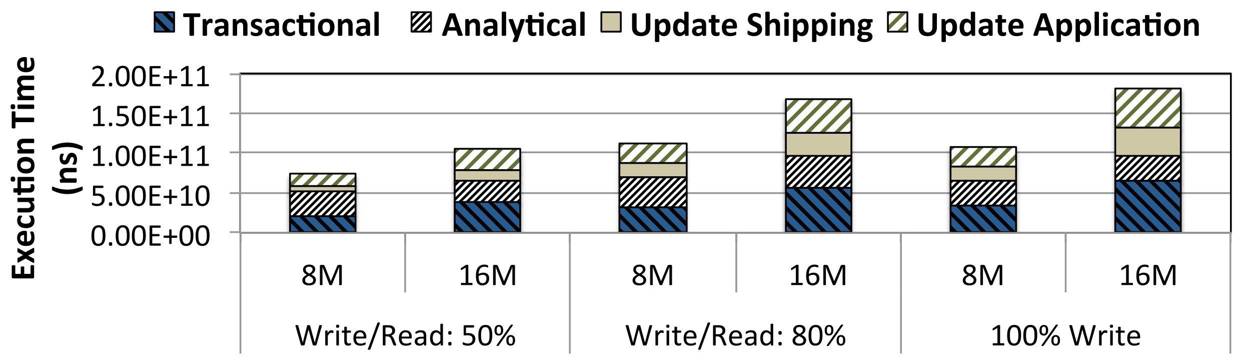


Figure 7.4. Execution time breakdown across different number of transactions and different write intensities.

Update Application: The update application process can be very challenging, due to the need to transform updates from one workload-specific format to another. For example, in relational analytics, the analytics engine uses several optimizations to speed up long-running scan queries and complex queries with multiple joins. To minimize the amount of data that needs to be accessed, analytics engines employ DSM representation to store data [348], and can compress tuples using an order-preserving dictionary-based compression (e.g., dictionary encoding [39, 238, 299]).

In our example, a single tuple update, stored in the NSM layout by the transactional workload, requires multiple random accesses to apply the update in the DSM layout. Compression further complicates this, as columns may need to be decompressed, updated, and recompressed. For compression algorithms that use sorted tuples, such as dictionary encoding, the updates can lead to expensive shifting of tuples as well. These operations generate a large amount of data movement and spend many CPU cycles. The challenges of update application are significant enough that some prior works give up on workload-specific optimization to try and maintain performance [255].

From Figure 7.3, we observe that the update application process reduces the transactional throughput of the Update-Propagation configuration by 49.6% compared tovs. zero-cost update propagation. As the write intensity increases (from 50% to 80%), the throughput suffers more (with a 59.0% drop at 80%). This is because 23.8% of the CPU cycles (and 30.8% of cache misses) go to the update application process (Figure 7.4), of which 62.6% is spent on (de)compressing columns.

Other Major Challenges. Like with single-instance design, we find that maintaining data consistency for multiple instances without compromising performance isolation is very challenging. Updates from transactions are frequently shipped and applied to analytics replicas while analytical queries run. As a result, multiple-instance-based systems suffer from the same consistency drawbacks that we observe for single-instance-based systems in Section 7.2.1. Another major challenge we find is the limited performance isolation. While separate instances provide partial performance isolation, as transactions and analytics do not compete for the same copy of data, they still share underlying hardware resources such as CPU cores and the memory system. As we discuss in Section 7.2.1, analytics workloads, as well as data freshness and consistency mechanisms, generate a large amount of data movement and take many cycles. As a result, multiple-instance designs also suffer from limited performance isolation.

We conclude that neither single- nor multiple-instance HTAP systems meet all of our desired HTAP properties. We therefore need a system that can avoid shared resource contention and alleviate the high data movement costs incurred for HTAP.

7.3. Polynesia

Our goal in this work is to design an HTAP system that can meet all of the desired HTAP properties, by avoiding the challenges that we identify for state-of-the-art HTAP systems in Section 7.2. To this end, we propose Polynesia which divides the HTAP system into multiple *islands*. Each island includes (1) a replica of data whose layout is optimized for a specific workload, (2) an optimized execution engine, and (3) a set of hardware resources (e.g., computation units, memory).resources. Polynesia has two types of islands: (1) a *transactional island*, and (2) an *analytical island*. To avoid the data movement and interference challenges that other multiple-instance-based HTAP systems face (see Section 7.2), we propose to equip each analytical island with (1) *in-memory hardware*; and (2) co-designed algorithms and hardware for the analytical execution engine, *update propagation*, and *consistency*.

Polynesia is a framework that can be applied to many different combinations of transactional and analytical workloads. In this work, we focus on designing an instance of Polynesia that supports relational transactional and analytical workloads.[1] Figure 7.5 shows the hardware for our chosen implementation, which includes one transactional island and one analytical island, and is equipped with a 3D-stacked memory similar to the Hybrid Memory Cube (HMC) [174], where multiple vertically-stacked DRAM cell layers are connected with a *logic layer* using thousands of *through-silicon vias* (TSVs). An HMC chip is split up into multiple *vaults*, where each vault corresponds to a vertical slice of the memory and logic layer. The transactional island uses an execution engine similar to conventional transactional engines [193, 385] to execute a relational transactional workload. The transactional island is equipped with conventional multicore CPUs and multi-level caches, as transactional queries have short execution times, are latency-sensitive, and have cache-friendly access patterns [43]. Inside each vault's portion of the logic layer in memory, we add hardware for the analytical island, including the update propagation mechanism (consisting of the *update shipping* and *update application* units), the consistency mechanism (*copy units*), and the analytical execution engine (simple programmable in-order PIM cores).In the next three sections,

[1]Note that our proposed techniques can be applied to other types of analytical workloads (e.g., graphs, machine learning) as well.

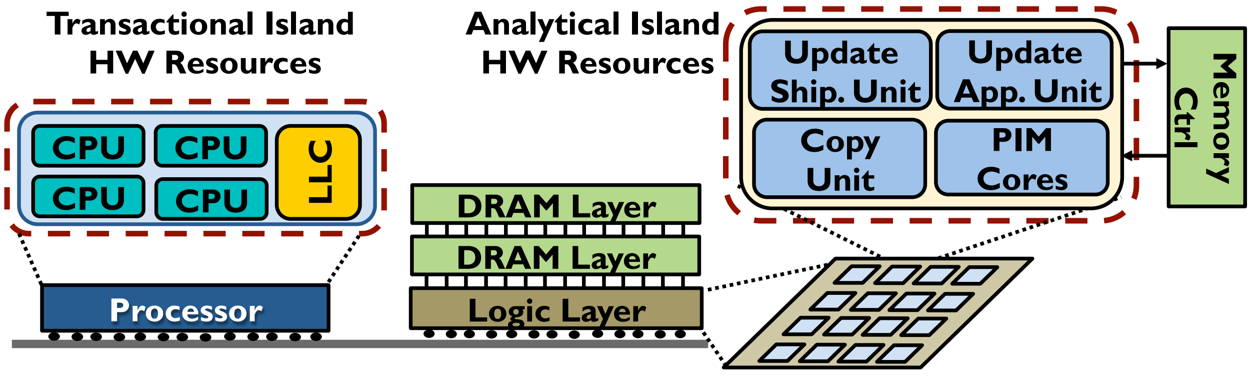


Figure 7.5. High-level organization of Polynesia hardware.

we discuss the detailed design of the update propagation mechanism (Section 7.4), consistency mechanism (Section 7.5), and analytical execution engine (Section 7.6).

To address potential capacity issues and accommodate larger data, Polynesia can extend across multiple memory stacks. We evaluate Polynesia with multiple stacks in Section 7.8.5.

7.4. Update Propagation Mechanism

We design a new two-part update propagation mechanism to overcome the high costs of analytical replica updates in state-of-the-art HTAP systems. The *update shipping unit* gathers updates from the transactional island, finds the target location in the analytical island, and frequently pushes these updates to the analytical island. The *update application unit* receives these updates, converts the updates from the transactional replica data format to the analytical replica data format, and applies the update to the analytical replica.

7.4.1. Update Shipping

Algorithm. Our update shipping mechanism includes three major stages. For each thread in the transactional engine, Polynesia stores an ordered *update log* for the queries performed by the thread. Each update log entry contains four fields: (1) a commit ID (a timestamp used to track the total order of all updates across threads), (2) the type of the update (insert, delete, modify), (3) the updated data, and (4) a record key (e.g., pair of row-ID and column-ID) that links this particular update to a column in the analytic replica. The update shipping process is triggered when total number of pending updates reaches the final log capacity, which we set to 1024 entries (see Section 7.4.2). The first stage is to scan the per-thread update logs, and merge them into a single *final log*, where all updates are sorted by the commit ID.

The second stage is to find the location of the corresponding column (in the analytical replica) associated with each update log entry. We observe that this stage is one of the major bottlenecks of update shipping, because the fields in each tuple in the transactional island are distributed across different columns in the analytical island. Since the column size is typically very large, finding the location of each update is a very time-consuming process. To overcome this, we maintain a hash index of data on the (column,row) key, and use that to find the corresponding column for each update in the final log. We use the modulo operation as the hash function. We size our hash table based on the column partition size. Similar to conventional analytical DBMSs, we can use soft partitioning [237, 255, 300] to address scalability issues when the column size increases beyond a certain point. Thus, the hash table size does not scale with column size. This stage contains a buffer for each column in the analytical island, and we add each update from the final log to its corresponding column buffer.

The final stage is to ship these buffers to each column in the analytical replica.

Hardware. We find that despite our best efforts to minimize overheads, our algorithm has three major bottlenecks that keep it from meeting data freshness and performance isolation requirements: (1) the scan and merge operation in stage 1, (2) hash index lookups in stage 2, and (3) transferring the column buffer contents to the analytical islands in stage 3. These primitives generate a large amount of data movement and account for 87.2% of our algorithm's execution time. To avoid these bottlenecks, we design a new hardware accelerator, called the *update shipping unit*, that speeds up the key primitives of the update shipping algorithm. We add this accelerator to each of Polynesia's in-memory analytical islands.

Figure 7.6 shows the high-level architecture of our in-memory update shipping unit. The update shipping unit consists of three building blocks: (1) a merge unit, (2) a hash lookup unit, and (3) a copy unit. The merge unit consists of 8 FIFO input queues, where each input queue corresponds to a sorted update log. Each input queue can hold up to 128 updates, which are streamed from DRAM.

The merge unit finds the oldest entry among the input queue heads, using a 3-level comparator tree, and adds it to the tail of the final log (a ninth FIFO queue). The final log is then sent to the hash unit to determine the target column for each update.

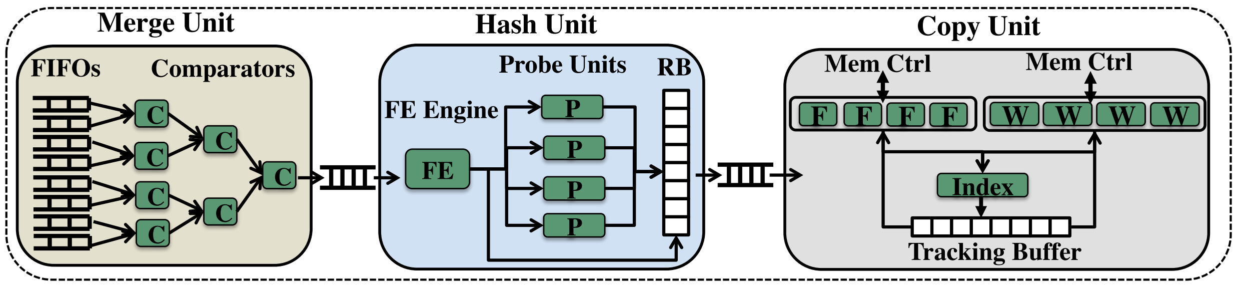


Figure 7.6. Update shipping unit architecture.

For our hash unit, we start with a preliminary design that includes a *probe unit*, a simple finite state machine controller that takes the address (for the key), computes the hash function to find the corresponding bucket in memory, and traverses the linked list of keys in the bucket. We find that having a single probe unit does not achieve our expected performance because (1) we cannot fully exploit the bandwidth of 3D-stacked memory; and (2) each lookup typically includes several pointer chasing operations that often leave the probe unit idle. As a result, we need to perform multiple hash lookups in parallel at each step. However, the challenge is that updates need to be handed to the copy unit in the same commit order they are inserted into the final update log

To address these challenges, we design the hash unit shown in Figure 7.6. Its key idea is to (1) decouple hash computation and bucket address generation from the actual bucket access/traversal, to allow for concurrent operations; and (2) use a small reorder buffer to track in-flight hash lookups, and maintain commit order for completed lookups that are sent to the copy engine. We introduce a front-end engine that fetches the keys from the final update log, computes the hash function, and sends the key address to probe units. The front-end engine allocates an entry (with the bucket address and a ready bit) for each lookup in the reorder buffer. We employ multiple probe units (we find that 4 strikes a balance between parallelism and area overhead), with each taking a bucket address and accessing DRAM to traverse the linked list.

We describe our copy engine in Section 7.5. Our analysis shows that the total area of our update shipping unit is 0.25 mm^2.

7.4.2. Update Application

Similar to other relational analytical DBMSs, our analytical engine uses the DSM data layout and *dictionary encoding* [39, 90, 92, 221, 238, 299, 348]. With dictionary encoding, each column in a table is transformed into a compressed column using encoded fixed-length integer values, and a dictionary stores a sorted mapping of real values to encoded values. As we discuss in Section 7.2.2, the layout conversion (our transactional island uses NSM) and column compression make update application process challenging. We design a new update application mechanism for Polynesia that uses hardware–software co-design to address these challenges.

Algorithm. We first discuss an initial algorithm that we develop for update application. We assume each column has n entries, and that we have m number of updates. First, the algorithm decompresses the encoded column by scanning the column and looking up in the dictionary to decode each item. This requires *n random accesses* to the dictionary. Second, the algorithm applies updates to the decoded column one by one. Third, it constructs a new dictionary, by sorting the updated column and calculating the number of fixed-length integer bits required to encode the sorted column. Dictionary construction is computationally expensive ($\mathscr{O}((n+m)\log(n+m))$) because we need to sort the entire column. Finally, the algorithm compresses the new column using our newly-constructednew dictionary. While entry decoding happens in constant time, encoding requires a logarithmic complexity search through the dictionary (since the dictionary is sorted).

This initial algorithm is memory intensive (Steps 1, 2, 4) and computationally expensive (Step 3). Having hardware support is *critical* to enable low-latency update application and performance isolation. While processing-in-memory (PIM)PIM may be able to help, our initial algorithm is not well-suited for PIM for two reasons, and we optimize the algorithm to address both.

Optimization 1: Two-Stage Dictionary Construction. We eliminate column sorting from Step 3, as it is computationally expensive. Prior work [301, 378] shows that to efficiently sort more than 1024 values in hardware, we should provide a hardware partitioner to split the values into multiple chunks, and then use a sorter unit to sort chunks one at a time. This requires an area of 1.13 mm^2 [301, 378]. Unfortunately, since tables can have millions of entries [221], we would need multiple sorter units to construct a new dictionary, easily exceeding the total area budget of 4.4 mm^2 per vault [42, 77, 99].

To eliminate column sorting, we sort only the dictionary, leveraging the fact that (1) the existing dictionary is already sorted, and (2) the new updates are limited to 1024 values. As a result, ourOur optimized algorithm initially builds a sorted dictionary for only the updates, which requires a single hardware sorter. A 1024-value bitonic sorter requires a much smaller area of only 0.18 mm^2 [378].sorter (a 1024-value bitonic sorter with an area of only 0.18 mm^2 [378]). Once the update dictionary is constructed, we now have two sorted dictionaries: the old dictionary and the update dictionary. We merge these into a single dictionary using a linear scan ($\mathcal{O}(n+m)$), and then calculate the number of bits required to encode the new dictionary.

Optimization 2: Reducing Random Accesses. To reduce the algorithm's memory intensity (which is a result of random lookups), we maintain a hash index that links the old encoded value in a column to the new encoded value. This avoids the need to decompress the column and add updates, eliminating data movement and random accesses for Steps 1 and 2, while reducing the number of dictionary lookups required for Step 4. The only remaining random accesses are for Step 4, which decrease from $\mathcal{O}((n+m)\log(n+m))$ to $\mathcal{O}(n+m)$.

We now describe our optimized algorithm. We first sort the updates to construct the update dictionary. We then merge the old dictionary and the update dictionary to construct the new dictionary and hash index. Finally, we use the index and the new dictionary to find the new encoded value for each entry in the column.

Hardware. We design a hardware implementation of our optimized algorithm, called the *update application unit*, and add it to each in-memory analytical island. The unit consists of three building blocks: a *sort unit*, a *hash lookup unit*, and a *scan/merge unit*. Our sort unit uses a 1024-value bitonic sorter, whose basic building block is a network of comparators. These comparators are used to form *bitonic sequences*, sequences where the first half of the sequence is monotonically increasing and the second half is decreasing. The hash lookup uses a simpler version of the component that we designed for update shipping. The simplified version does not use a reorder buffer, as there is no dependency between hash lookups for update application. We use the same number of hash units (empirically set to 4), each corresponding to one index structure, to parallelize the compression process. For the merge unit, we use a similar design from our update shipping unit. Our analysis shows that the total area of our update application unit is 0.4 mm^2.

7.5. Consistency Mechanism

We design a new consistency mechanism for Polynesia in order to deliver all of the desired properties of an HTAP system (Section 7.2). Our consistency mechanism must not compromise either the throughput of analytical queries or the rate at which updates are applied. This sets two requirements for our mechanism: (1) analytics must be able to run all of the time without slowdowns, to satisfy the performance isolation property; and (2) the update application process should not be blocked by long-running analytical queries, to satisfy the data freshness property. This means that our mechanism needs a way to allow analytical queries to run concurrently with updates, without incurring the long chain read overheads of similar mechanisms such as MVCC (see Section 7.2.1).

Algorithm. Our mechanism relies on a combination of snapshotting [205] and versioning [37] to provide snapshot isolation [36, 47] for analytics. Our consistency mechanism is based on two key observations: (1) updates are applied at a column granularity, and (2) snapshotting a column is cost-effective using PIM logic. We assume that for each column, there is a chain of snapshots where each chain entry corresponds to a version of this column. Unlike chains in MVCC, each version is associated with a column, not a tuple.

We adopt a lazy approach (late materialization [8]), where Polynesia does not create a snapshot every time a column is updated. Instead, on a column update, Polynesia marks the column as dirty, indicating that the snapshot chain does not contain the most recent version of the column data. When an analytical query arrives Polynesia checks the column metadata, and creates a new snapshot only if (1) any of the columns are dirty (similar to Hyper [205]), and (2) no current snapshot exists for the same column (we let multiple queries share a single snapshot). During snapshotting, Polynesia updates the head of the snapshot chain with the new value, and marks the column as clean. This provides two benefits. First, the analytical query avoids the chain traversals and timestamp comparisons performed in MVCC, as the query only needs to access the head of the chain at the time of the snapshot. Second, Polynesia uses a simple yet efficient garbage collection: when an analytical query finishes, snapshots no longer in use by any query are deleted (aside from the head of the chain).

To guarantee high data freshness (*second requirement*), our consistency mechanism always lets transactional updates directly update the main replica using our two-phase update application algorithm (Section 7.4.2). In Phase 1, the algorithm constructs a new dictionary and a new column. In Phase 2, the algorithm atomically updates the main replica with pointers to the new column and dictionary.

Hardware. Our algorithm's success at satisfying the first requirement for a consistency mechanism (i.e., no slowdown for analytics) relies heavily on its ability to perform fast memory copies to minimize the snapshotting latency. Therefore, we add a custom copy unit to each of Polynesia's in-memory analytical islands. We have two design goals for the unit. First, the accelerator needs to be able to issue multiple memory accesses concurrently. This is because (1) we are designing the copy engine for an arbitrarily-sized memory region (e.g., a column), which is often larger than the memory access granularity per vault (8–16B) in an HMC-like memory; and (2) we want to fully exploit the internal bandwidth of 3D-stacked memory. Second, when a read for a copy completes, the accelerator should immediately initiate the write.

We design our copy unit (Figure 7.6) to satisfy both design goals. To issue multiple memory accesses concurrently, we leverage the observation that these memory accesses are independent. We use multiple fetch and writeback units, which can read from or write to source/destination regions in parallel. To satisfy the second design goal, we need to track outstanding reads, as they may come back from memory out of order. Similar to prior work on accelerating `memcpy` [256], we use a *tracking buffer* in our copy unit. The buffer allocates an entry for each read issued to memory, where an entry contains a memory address and a ready bit. Once a read completes, we find its corresponding entry in the buffer and set its ready bit to trigger the write.

We find that the buffer lookup limits the performance of the copy unit, as each lookup results in a full buffer scan, and multiple fetch units perform lookups concurrently (generating high contention). To alleviate this, we design a hash index based on the memory address to determine the location of a read in the buffer. We make use a similar design as the hash lookup unit in our update shipping unit.

7.6. Analytical Engine

The analytical execution engine performs the analytical queries. When a query arrives, the engine parses the query and generates an algebraic query plan consisting of physical operators (e.g., scan, filter, join). In the query plan, operators are arranged in a tree where data flows from the bottom nodes (leaves) toward the root, and the result of the query is stored in the root. The analytical execution engine employs the top-down Volcano (Iterator) execution model [133, 278] to traverse the tree and execute operators while respecting dependencies between operators. Analytical queries typically exhibit a high degree of both intra- and inter-query parallelism [237, 300, 388]. To exploit this, the engine decomposes a query into multiple tasks, each being a sequence of one or more operators. The engine (task scheduler) then schedules the tasks, often in a way that executes multiple independent tasks in parallel.

Efficient analytical query execution strongly depends on (1) data layout and data placement, (2) the task scheduling policy, and (3) how each physical operator is executed. Like prior works [77, 380], we find that the execution of physical operators of analytical queries significantly benefit from PIM. However, without a HTAP-aware and PIM-aware data placement strategy and task scheduler, PIM logic for operators alone cannot provide significant throughput improvements.

We design a new analytical execution engine based on the characteristics of our in-memory hardware. As we discuss in Section 7.3, Polynesia uses a 3D-stacked memory that contains multiple vaults. Each vault (1) provides only a fraction (e.g., 8 GB/s) of the total bandwidth available in a 3D-stacked memory, (2) has limited power and area budgets for PIM logic, and (3) can access its own data faster than it can access data stored in other vaults (which take place through a vault-to-vault interconnect). We take these limitations into account as we design our data placement mechanism and task scheduler.

7.6.1. Data Placement

We evaluate three different data placement strategies for Polynesia. Our analytical engine uses the DSM layout to store data, and makes use of dictionary encoding [39] for column compression. Our three strategies affect which vaults the compressed DSM columns and dictionary are stored in.

Strategy 1: Store the Entire Column (with Dictionary) in One Vault. This strategy has two major benefits. First, both the dictionary lookup and column access are to the local vault, which improves analytical query throughput. Second, it simplifies the update application process (Section 7.4.2), as the lack of remote accesses avoids the need to synchronize updates between multiple update applications units (as each vault has its own unit).

However, this data placement strategy forces us to service tasks that access a particular column using (1) highly-constrained PIM logic (given the budget of a single vault), and (2) only the bandwidth available to one vault. This significantly degrades throughput, as it becomes challenging to benefit from intra-query parallelism with only one highly-constrained set of logic available for a query.

Strategy 2: Partition Columns Across All Vaults in a Chip. To address the challenges of Strategy 1, we can partition each column and distribute it across all of the vaults in the 3D-stacked memory chip (i.e., a *cube*). This approach allows us to (1) exploit the entire internal bandwidth of the 3D-stacked memory, and (2) use all of the available PIM logic to service each query. Note that unlike partitioning the column, partitioning the dictionary across all vaults is challenging because the dictionary is sorted, forcing us to scan the entire column and find the corresponding dictionary entries for each column entry.

However, Strategy 2 suffers from two major drawbacks. First, it makes the update application (Section 7.4.2) significantly challenging. To perform update application under Strategy 2, we need to (1) perform many remote accesses to gather all of the column partitions, (2) update the column, and then (3) perform many remote accesses to scatter the updated column partitions back across the vaults. Given the high frequency of update application in HTAP workloads, this gather/scatter significantly increases the latency of update application and intra-cube traffic. Second, we need to perform several remote accesses to collect sub-results from each partition and aggregate them, which reduces the throughput.

Strategy 3: Partition Columns Across a Group of Vaults. To overcome the challenges of Strategies 1 and 2, we propose a hybrid strategy where we create small *vault groups*, consisting of a fixed number of vaults, and partition a column across the vaults in a vault group. For a group with v vaults, this allows us to increase the aggregate bandwidth for servicing each query by v times, and provides up to v times the power and area for PIM logic. The number of vaults per group is critical for efficiency: too many vaults can complicate the update application process, while not enough vaults can degrade throughput. We empirically find that four vaults per group strikes a good balance.

While the hybrid strategy reduces the cost of update application compared to Strategy 2, it still needs to perform remote accesses within each vault group. To overcome this, we leverage an observation from prior work [221] that the majority of columns have only a limited (up to 32) number of distinct values. This means that the entire dictionary incurs negligible storage overhead (˜2 KB). To avoid remote dictionary accesses during update application, Strategy 3 keeps a copy of the dictionary in each vault. Such an approach is significantly costlier under Strategy 2, as for a given column size, the number of dictionary copies scales linearly with the number of column partitions, which is particularly problematic in a system with multiple memory stacks.

Polynesia makes use of Strategy 3 for data placement.

7.6.2. Scheduler

Polynesia's task scheduler plays a key role in (1) exploiting inter- and intra-query parallelism, and (2) efficiently utilizing hardware resources. For each query, the scheduler (1) decides how many tasks to create, (2) finds how to map these tasks to the available hardware resources (*PIM threads*), and (3) guarantees that dependent tasks are executed in order. We first design a basic scheduler heuristic that generates tasks (statically at compile time) by disassembling the operators of the query plan into operator instances (i.e., an invocation of a physical operator on some subset of the input tuples) based on (1) which vault groups the input tuples reside in; and (2) the number of available PIM threads in each vault group, which determines the number of tasks generated. The scheduler inserts tasks into a global work queue in an order that preserves dependencies between operators, monitors the progress of PIM threads, and assigns each task to a free thread (push-based assignment).

However, we find that this heuristic is not optimized for PIM, and leads to sub-optimal performance due to three reasons. First, the heuristic requires a dedicated runtime component to monitor and assign tasks. The runtime component must be executed on a general-purpose PIM core, either requiring another core (difficult given limited area/power budgets) or preempting a PIM thread on an existing core (which hurts performance). Second, the heuristic's static mapping is limited to using only the resources available within a single vault group, which can lead to performance issues for queries that operate on very large columns. Third, this heuristic is vulnerable to load imbalance, as some PIM threads might finish their tasks sooner and wait idly for straggling threads.

We optimize our heuristic to address these challenges. First, we design a pull-based task assignment strategy, where PIM threads cooperatively pull tasks from the task queue at runtime. This eliminates the need for a runtime component (first challenge) and allows PIM thread to dynamically load balance (third challenge). To this end, we introduce a local task queue for each vault group. Each PIM thread looks into its own local task queue to retrieve its next task. Second, we optimize the heuristic to allow for finer-grained tasks. Instead of mapping tasks statically, we partition input tuples into fixed-size segments (i.e., 1000 tuples) and create an operator instance for each partition. The scheduler then generates tasks for these operator instances and inserts them into corresponding task queues (where those tuple segments reside). The greater number of tasks increases opportunities for load balancing. Finally, we optimize the heuristic to allow a PIM thread to steal tasks from a remote vault if its local queue is empty. This enables us to potentially use all available PIM threads to execute tasks, regardless of the data distribution (second challenge). Each PIM thread first attempts to steal tasks from other PIM threads in its own vault group, because the thread already has a local copy of the full dictionary in its vault, and needs remote vault accesses only for the column partition. If there is no task to steal in its vault group, the PIM thread attempts to steal a task from a remote vault group.

7.6.3. Hardware Design

Given area and power constraints, it can be difficult to add enough PIM logic to each vault to saturate the available vault bandwidth [77]. Mondrian [77] attempts to change the access pattern from random to sequential, allowing PIM threads to use stream buffers to increase bandwidth utilization. With our new data placement strategy and scheduler, we instead expose greater intra-query parallelism, and use simple programmable in-order PIM cores to exploit the available vault bandwidth. We add four PIM cores to each vault, where the cores are similar to those in prior work [77, 97]. We run a PIM thread on each core, and we use these cores to execute the scheduler and other parts of the analytical engine (e.g., query parser).

We find that our optimized heuristic significantly increases data sharing between PIM threads. This is because within each vault group, all 16 PIM threads access the same local task queue, and must synchronize their accesses. The problem worsens when other PIM threads attempt to steal tasks from remote vault groups, especially for highly-skewed workloads. To avoid excessive accesses to DRAM and let PIM threads share data efficiently, we implement a simple fine-grained coherence technique, which uses a local PIM-side directory in the logic layer to implement a low-overhead coherence protocol.

The PIM-side directory is also used to maintain coherence between PIM cores (or PIM accelerators) and the CPUs. This is because to allow coordination, ordering, and synchronization between different parts of islands, we need to provide coherence between CPU cores and PIM logic. We employ a simple fine grained coherence technique, which uses the local PIM-side directory in the logic layer to enable low-overhead fine-grained coherence between PIM logic and the CPUs. The CPU-side directory acts as the main coherence point for the system, interfacing with both the processor caches and the PIM-side directory.

7.7. Methodology

We use and heavily extend state-of-the-art transactional and analytical engines to implement various single- and multiple-instance HTAP configurations. We use DBx1000 [385, 386] as the starting point for our transactional engine. The transactional engine spawns multiple worker threads, where each thread is mapped to a CPU core and executes one transactional query. We implement an in-house analytical engine similar to C-store [348], with each analytical worker thread mapped to a CPU core and executing a single task. The analytical engine supports key physical operators for relational analytics: select, filter, aggregate and join. For join, we use a state-of-the-art, highly-optimized hash join kernel [358]. The analytical engine supports both NSM and DSM layouts, and dictionary encoding [39, 238, 299]. For consistency, we implement both snapshotting (similar to software snapshotting [368], and with snapshots taken only when dirty data exists) and MVCC (adopted from DBx1000 [385]).

Our baseline single-instance HTAP system stores the single data replica in main memory. The system consists of 16 tables, 256K tuples per table, 4 randomly-populated integer fields, and 4 randomly-populated character fields per table, with tables in the NSM format. Each transactional query randomly performs reads or writes on a few randomly-chosen tuples from a randomly-chosen table. Each analytical query uses select and join on randomly-chosen tables and columns. We evaluate both snapshotting and MVCC for single-instance. Our baseline multiple-instance HTAP system models a similar system as our single-instance baseline, but provides the transactional and analytical engines with separate replicas (using the NSM layout for transactions, and DSM with dictionary encoding for analytics). The system uses the mechanism we describe in Section 7.4 to propagate updates. Across all baselines, we have 4 transactional and 4 analytical worker threads. We use hardware performance counters to analyze each baseline on real systems.

We simulate Polynesiaand compare it to the baselines using gem5 [38]. We perform all simulations in full-system mode using the x86 ISA, and modify the integrated DRAMSim2 [75] DRAM simulator to model an HMC-like 3D-stacked DRAM [174]. Table 7.1 shows our system configuration. For the analytical island, each vault of our 3D-stacked memory contains four PIM cores and three fixed-function accelerators (update shipping unit, update application unit, copy unit). For the PIM core, we model a core similar to the ARM Cortex-A7 [27].

Area. Our four PIM cores require 1.8 mm^2, based on the Cortex-A7 (0.45 mm^2 each) [27]. We use Calypto Catapult to determine the area of the accelerators for a 22nm process: 0.7 mm^2 for the

Processor (Transactional Island)	4 OoO cores, each with 2 HW threads, 8-wide issue; *L1 I/D Caches*: 64 kB private, 4-way assoc.; *L2 Cache:* 8 MB shared, 8-way assoc.; *Coherence*: MESI
PIM Core	4 in-order cores per vault, 2-wide issue, *L1 I/D Caches*: 32 kB private, 4-way assoc.
3D-Stacked Memory	4 GB cube, 16 vaults per cube; *Internal Bandwidth:* 256 GB/s; *Off-Chip Channel Bandwidth:* 32 GB/s

Table 7.1. Evaluated system configuration for Polynesia.

update propagation units and 0.2 mm^2 for the in-memory copy unit for our consistency mechanism. This brings Polynesia's total to 2.7 mm^2 per vault.

Energy. We model system energy similar to prior work [42, 43], which sums the energy consumed by the CPU cores, DRAM, on-chip and off-chip interconnects, and all caches. We leverage estimates and models of 3D-stacked DRAM energy from prior work [184]. We estimate the energy consumption of all caches using CACTI-P 6.5 [270], assuming a 22 nm process. We model the off-chip interconnect using the method used by prior work [97], which estimates the HMC SerDes energy consumption as 3 pJ/bit for data packets. We estimate the CPU energy based on prior works [42, 290, 364] and scale the energy to accurately fit our processor. We used prior work's energy model [42] to estimate the energy consumed by the PIM core and specialized accelerators.

7.8. Evaluation

We first evaluate the three major components of our proposal: (1) update propagation, (2) our consistency mechanism, and (3) our analytical engine. We then perform an end-to-end system evaluation.

7.8.1. End-to-End System Analysis

Figure 7.7 (left) shows the transactional throughput for six DBMSs: (1) Single-Instance-Snapshot (*SI-SS*); (2) Single-Instance-MVCC (*SI-MVCC*); (3) *MI+SW*, an improved version of Multiple-Instance that includes all of our software optimizations for Polynesia (except those specifically targeted for PIM); (4) *MI+SW+HB*, a hypothetical version of MI+SW with 8x bandwidth (256 GB/s), equal to the internal bandwidth of HBM; (5) *PIM-Only*, a hypothetical version of MI+SW which uses general-purpose PIM cores to run both transactional and analytical workloads;

and (6) *Polynesia*, our full hardware–software proposal. We normalize throughput to an ideal transaction-only DBMS (*Ideal-Txn*) for each transaction count. Of the single-instance DBMSs, SI-MVCC performs best, coming within 20.0% of the throughput of Ideal-Txn on average. Its use of MVCC over snapshotting overcomes the high performance penalties incurred by SI-SS. For the two software-only multiple instance DBMSs (MI+SW and MI+SW+HB), despite our enhancements, both fall significantly short of SI-MVCC due to their lack of performance isolation and, in the case of MI+SW, the overhead of update propagation. MI+SW+HB, even with its higher available bandwidth, cannot data movement or contention on shared resources. As a result, its transactional throughput is still 41.2% lower than Ideal-Txn. PIM-only significantly hurts transactional throughput (by 67.6% vs. Ideal-Txn), and even performs 7.6% worse than SI-SS. Polynesia improves the average throughput by 51.0% over MI+SW+HB, and by 14.6% over SI-MVCC, because it (1) uses custom PIM logic for analytics along with its update propagation and consistency mechanisms to significantly reduce contention, and (2) reduces off-chip bandwidth contention by reducing data movement. As a result, Polynesia comes within 8.4% of Ideal-Txn.

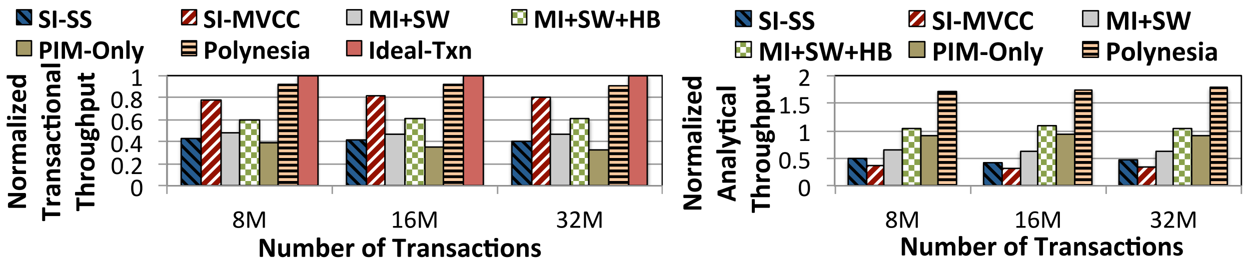


Figure 7.7. Normalized transactional (left) and analytical (right) throughput for end-to-end HTAP systems.

Figure 7.7 (right) shows the analytical throughput across the same DBMSs. We normalize throughput at each transaction count to a baseline where analytics are running alone on our multicore system. We see that while SI-MVCC is the best software-only DBMS for transactional throughput, it degrades analytical throughput by 63.2% compared to the analytics baseline, due to its lack of workload-specific optimizations and poor consistency mechanism (MVCC). Neither of these problems can be addressed by providing higher bandwidth. MI+SW+HB is the best software-only HTAP DBMS for analytics, because it provides workload-specific optimizations, but it still loses 35.3% of the analytical throughput of the baseline. MI+SW+HB improves throughput by 41.2% over MI+SW but still suffers from resource contention due to update propagation and the consistency mechanism. PIM-Only performs similar to MI+SW+HB but reduces throughput by 11.4% compared to that, as it suffers from resource contention caused by co-running transactional queries. Polynesia *improves* over the baseline by 63.8%, by eliminating data movement, having low-latency accesses, and using custom logic for update propagation and consistency.

Overall, averaged across all transaction counts in Figure 7.7, Polynesia has a higher transactional throughput (2.20X over SI-SS, 1.15X over SI-MVCC, and 1.94X over MI+SW; mean of 1.70X), *and* a higher analytical throughput (3.78X over SI-SS, 5.04X over SI-MVCC, and 2.76X over MI+SW; mean of 3.74X).

Real Workload Analysis. To model more complex queries, we evaluate Polynesia using a mixed workload from TPC-C [2] (for our transactional workload) and TPC-H [3] (for our analytical workload). TPC-C's schema includes nine relations (tables) that simulate an order processing application. We simulate two transaction types defined in TPC-C, *Payment* and *New order*, which together account for 88% of the TPC-C workload [385]. We vary the number of warehouses from 1 to 4, and we assume that our transactional workload includes an equal number of transactions from both *Payment* and *New order*. TPC-H's schema consists of eight separate tables We use TPC-H Queries 1–6, long and complex workloads that perform selection, aggregation, hash join, and sorting over the *Lineitem* (with 6 million rows), *Orders* (with 1.5 million rows), and *Customers* (150 thousand rows) Tables.

We evaluate the transactional and analytical throughput for Polynesia and for three baselines: (1) SI-SS, (2) SI-MVCC, (3) MI+SW (results not shown). We find that, averaged across all warehouse counts, Polynesia has a higher transactional throughput (2.05X over SI-SS, 1.23X over SI-MVCC, and 1.63X over MI+SW; mean of 1.63X), and a higher analytical throughput (3.6X over SI-SS, 4.54X over SI-MVCC, and 2.37X over MI+SW; mean of 3.50X) over all three baselines.

We conclude that Polynesia's ability to meet all three HTAP properties enables better transactional and analytical performance over all three of our state-of-the-art systems. In Sections 7.8.2, 7.8.3, and 7.8.4, we study how each component of Polynesia contributes to performance.

7.8.2. Update Propagation

Figure 7.8 shows the transactional throughput for Polynesia's update propagation mechanism and Multiple-Instance, normalized to a multiple-instance baseline with zero cost for update propagation (*Ideal*). We assume each analytical worker thread executes 128 queries, and vary both the number of transactional queries per worker thread and the transactional query read-to-write ratio. To isolate the impact of different update propagation mechanisms, we use a zero-cost consistency mechanism, and ensure that the level of interference remains the same for all mechanisms.

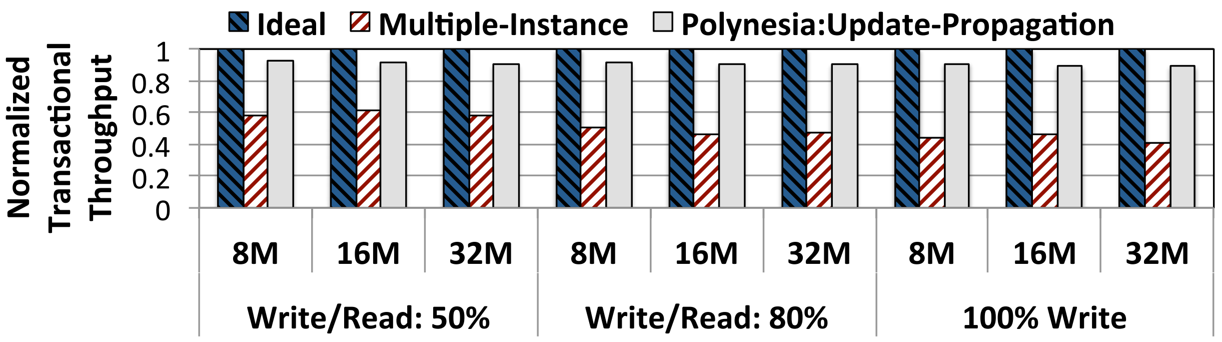


Figure 7.8. Effect of update propagation mechanisms on transactional throughput.

We find that Multiple-Instance degrades transactional throughput on average by 49.5% compared to Ideal, as it severely suffers from resource contention and data movement cost. 27.7% of the throughput degradation comes from the update shipping latencies associated with data movement and with merging updates from multiple transactional threads together. The remaining degradation is due to the update application process, where the major bottlenecks are column compression/de-compression and dictionary reconstruction. Our update propagation mechanism, on the other hand, improves throughput by 1.8X compared to Multiple-Instance, and comes within 9.2% of Ideal. The improvement comes from (1) significantly reducing data movement by offloading update propagation process to PIM, (2) freeing up CPUs from performing update propagation by using a specialized hardware accelerator, and (3) tuning both hardware and software. In all, our mechanism reduces the latency of update propagation by 1.9X compared to Multiple-Instance (not shown). We conclude that our update propagation mechanism provides data freshness (i.e., low update latency) while maintaining high transactional throughput (i.e., performance isolation).

7.8.3. Consistency Mechanism

Figure 7.9 (left) shows the analytical throughput of Polynesia's consistency mechanism and of Single-Instance-MVCC (*MVCC*), normalized to a single-instance baseline with zero cost for MVCC (*Zero-Cost-MVCC*) for each query count. We assume each analytical worker thread executes 128 queries, and we vary the transactional query count per worker thread. For a fair comparison, we implement our consistency mechanism in a single-instance system. We find that MVCC degrades analytical throughput, on average, by 37.0% compared to Ideal-MVCC, as it forces each analytical query to traverse a lengthy version chain and perform expensive timestamp comparisons to locate the most recent version. Our consistency mechanism, on the other hand, improves analytical throughput by 1.4X compared to MVCC, and comes within 11.7% of Ideal-MVCC, because it does not force analytical queries to scan lengthy version chains when accessing each tuple.

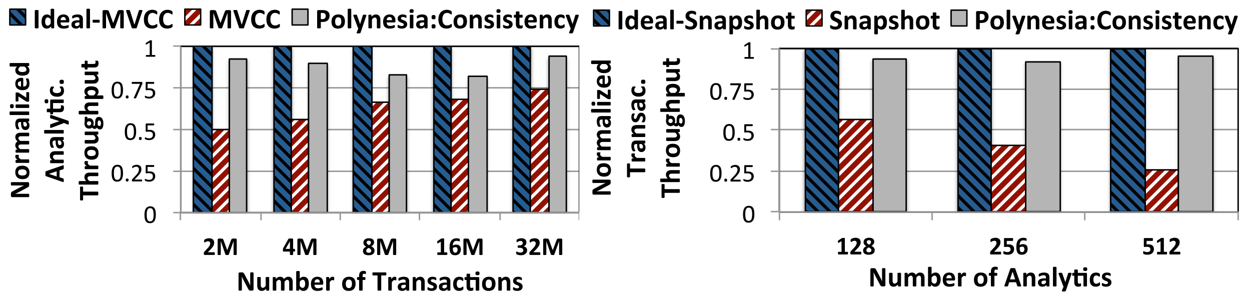


Figure 7.9. Effect of consistency mechanisms on analytical (left) and transactional (right) throughput.

Figure 7.9 (right) shows the transactional throughput for Polynesia's consistency mechanism and Single-Instance-Snapshot (*Snapshot*), normalized for each workload count to a single-instance baseline with zero-cost snapshotting (*Ideal-Snapshot*). Each worker thread performs 1M transactional queries, and we vary the number of analytical queries. We find that Snapshot reduces transactional throughput, on average, by 59% compared to Ideal-Snapshot. This is because of expensive `memcpy` operations needed to create each snapshot, resulting in significant resource contention Polynesia's mechanism improves transactional throughput by 2.2X over Snapshot, and comes within 6.1% of Ideal-Snapshot, because it performs snapshottingsnapshots at a column granularity and leverages PIM to performfor fast snapshotting.

We conclude that our consistency mechanism maintains consistency without compromising performance isolation.

7.8.4. Analytical Engine

We study the effect of each of our data placement strategies from Section 7.6.1: Strategy 1 (*Local*), Strategy 2 (*Remote*), our *Hybrid* strategy, where all use the basic scheduler heuristic. We also study our hybrid strategy combined with our improved scheduler heuristic (labeled as *Hybrid+sched*). For these studies, we send all analytical queries to the same column.

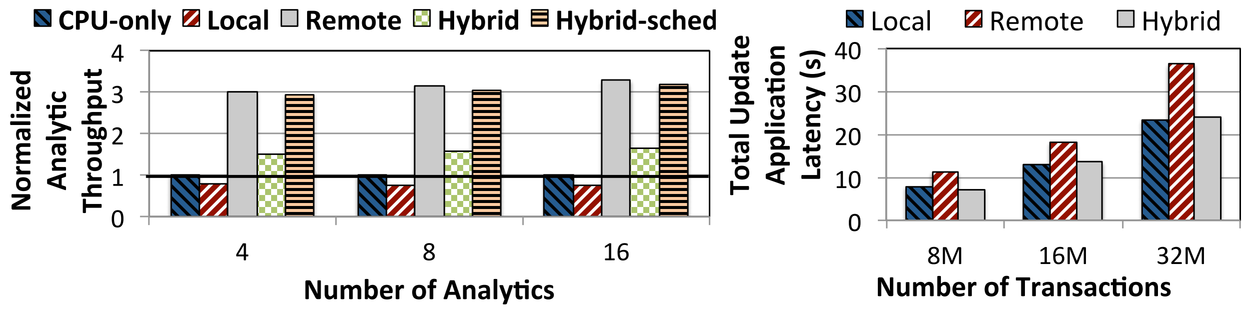


Figure 7.10. Normalized analytical throughput (left) and update application latency (right) across different data placement and task scheduling strategies.Effect of data placement/scheduling on throughput (left) and update application latency (right).

Figure 7.10 (left) shows the analytical throughput, normalized to the CPU-only baseline for each analytic workload count, where one core services all queries to the same column. We find that Local reduces throughput by 23.9% on average over CPU-only, because in Local, each analytical query can only use (1) the PIM cores in the local vault, which cannot issue many memory requests concurrently, and (2) a single vault's bandwidth. In contrast, CPU-only leverages the out-of-order cores to issue many memory requests in parallel. Remote improves throughput by 4.1X/3.1X over Local/CPU-only. This is because under Remote, each column is partitioned across all of the vaults, allowing us to service each query using (1) all of the PIM cores, and (2) the entire internal bandwidth of the memory. However, Remote increases the update application latency, on average, by 45.8% (Figure 7.10 (right)), and thus, degrades data freshness. This is because of the high update application costs that we discuss in Section 7.6.1, which Local does not incur.

We find that Hybrid addresses the shortcomings of Local, improving throughput by 57.2% over

CPU-only, while having a similar update application latency (0.7ms). This is because the local dictionary copies eliminate most of the remote accesses. However, the throughput under Hybrid is 49.8% lower than Remote, because each query is serviced only using resources (bandwidth and computation) available in the local vault group. Hybrid-sched overcomes this limitation thanks to task stealing, making idle resources in remote vaults available for analytical queries, and comes within 3.2% of Remote, while maintaining the same update application latency as Hybrid. Note that Remote's slightly higher throughput than Hybrid-sched is because in Hybrid-sched, every memory access for a task stolen from another vault group is remote.

Comparison to Mondrian. We evaluate the performance benefit of Polynesia's analytical engine over Mondrian [77]. Our analysis shows that Polynesia's analytical engine performs on average 63.2% better than Mondrian in terms of analytical throughput. The reason is that Mondrian accelerates only physical operators. Our scheduling policy and data placement enables us to efficiently execute analytical queries and achieve higher analytical throughput than Mondrian. Note that simply extending the Mondrian engine (or any prior PIM-based proposal for analytical workload) to support transactional workload does not address the key HTAP challenges, as we still need to provide data propagation and consistency mechanisms.

7.8.5. Multiple Memory Stacks

Figure 7.11 (left) shows how Polynesia performs as the dataset size grows. To accommodate the larger data, we increase the number of HMC stacks, doubling the data set size as we double the stack count. In these studies, we use a workload with 32M transactional and 60K analytical queries, and analyze analytical throughput normalized to Multiple-Instance (MI) as a case study. We assume stacks are connected together using a processor-centric topology [43]. To provide a fair comparison, we double the number of cores available to the analytical threads in the MI baseline as we double the number of stacks, to compensate for the doubling of hardware resources available to Polynesia (since there are twice as many vaults). We find that Polynesia significantly outperforms MI (up to 3.0X) and scales well as we increase the stack count. This is because, as we increase stack count, columns can be distributed more evenly across vault groups, which reduces the probability of

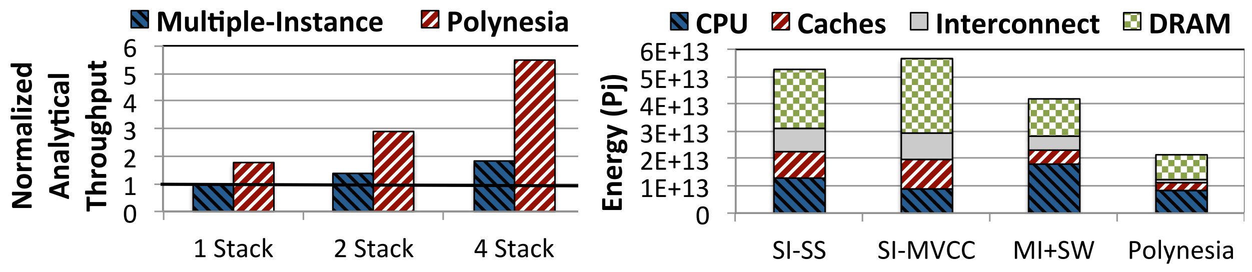


Figure 7.11. Effect of increasing dataset size on analytical throughput (left). Total system energy (right).

multiple queries colliding in the same vault group. On the other hand, with increasing dataset size, the overheads of consistency mechanism, update propagation and analytical query execution are all higher for MI, which hurts its scalability. The transactional throughput (not shown) decreases by 54.4% at four stacks for MI, compared to one stack, but decreases by only 8.8% for Polynesia.

7.8.6. Energy Analysis

Figure 7.11 (right) shows the total system energy across different HTAP DBMSs. We find that MI+SW performs better than SI-MVCC and SI-SS in terms of energy consumption, but still uses a large amount of energy due to a large number of accesses to off-chip memory, large caches, and using power-hungry CPU cores. These challenges cannot be solved by providing high bandwidth to CPU cores. Polynesia eliminates a significant amount of off-chip accesses, and uses custom logic and simple in-order PIM cores, reducing its energy consumption by 48% over MI+SW.

7.9. Summary

We propose Mensa, a novel HTAP system that makes use of multiple workload-optimizedworkload-optimized transactional and analytical islands to enable real-time analytics without sacrificing throughput. In Mensa, transactional islands make use of conventional transactional database engines running on multicore CPUs, while analytical islands make use of custom co-designed algorithms and hardware that are placed inside memory. Our analytical islands are designed to alleviate the data movement and workload interference costs incurred in state-of-the-art HTAP systems, while still ensuring that data replicas for analytics workloads are kept up-to-date with the most recent version of the transactional data replicas. Mensa outperforms three state-of-the-art HTAP systems (with a 1.7X/3.7X higher transactional/analytical throughput on average), while consuming less energy (48% lower than the best).

Chapter 8

Conclusion

8.1. Summary of the Dissertation

Many modern and emerging applications (e.g., deep learning, data analytics, machine learning, IoT) must process increasingly large volumes of data. Unfortunately, for these modern and emerging applications, the large amounts of data that need to move across the memory channel create a large data movement bottleneck in the computing system. One potential solution to mitigate this bottleneck is PIM, where we avoid or reduce unnecessary data movement by executing the data-movement-heavy portions of our applications close to the data. While PIM can allow many data-intensive applications to avoid moving data from memory to the CPU, there are many practical system-level challenges that need to be solved to enable the widespread adoption of PIM. Our goal in this thesis is to make PIM effective and practical in conventional computing systems. Toward this end, we propose a series of practical mechanisms to facilitate the systematic offloading of computation to PIM logic and reduce processor–memory data movement in modern workloads.

First, we analyze several widely-used Google consumer workloads to examine the suitability of PIM across key workloads in Chapter 4. We find that data movement contributes to a significant portion (62.7%) of their total energy consumption. Our analysis reveals that the majority of this data movement comes from a number of simple functions and primitives that are good candidates to be executed on low-power processing-in-memory (PIM) logic. We then comprehensively study the energy and performance benefit of PIM to address the data movement cost on Google consumer workloads. Our evaluation shows that offloading simple functions from these consumer workloads to PIM logic, consisting of either simple cores or specialized accelerators, reduces system energy consumption by 55.4% and execution time by 54.2%, on average across all of our workloads.

Second, in Chapter 5, we address the coherence challenge for PIM/NDAs, which is one of the major system challenges for adopting PIM in computing systems. We extensively analyze NDA applications and existing coherence mechanisms, and observe that (1) a majority of off-chip coherence traffic is unnecessary, and (2) a significant portion of off-chip data movement can be eliminated if a coherence mechanism has insight into NDA memory accesses. Based on our observations, we propose CoNDA, a coherence mechanism that lets an NDA optimistically execute code assuming that it has coherence permissions. Optimistic execution enables CoNDA to gather information on memory accesses, and exploit the information to minimize unnecessary off-chip data movement for coherence. Our results show that CoNDA improves performance and reduces energy consumption compared to existing coherence mechanisms, and comes close to the energy and performance of a no-cost ideal coherence mechanism.

Third, in Chapter 6, we propose a hardware–software co-design approach aware of PIM for mobile machine learning applications to enable energy-efficient and high-performance inference execution. We conduct the first bottleneck analysis of the Google Edge TPU, a state-of-the-art ML inference accelerator, as it executes 24 state-of-the-art Google edge NN models. Our analysis reveals that the Edge TPU's monolithic design leads to significant underutilization and poor energy efficiency for our edge NN models, which exhibit significant layer variation. We propose a new framework called Mensa, consisting of multiple small heterogeneous accelerators, each specialized to specific layer characteristics. Using our discovery that layers group into a small number of clusters, we create a Mensa design for the Google edge NN models consisting of three accelerators. Compared to the Edge TPU and to a state-of-the-art reconfigurable ML accelerator, our design improves energy efficiency by 3.0x/2.4x and throughput by 3.1x/4.3x for our edge NN models.

Finally, we study another emergying and modern application (HTAP systems) in Chapter 7 and propose a new hardware–software cooperative design for HTAP databases aware of PIM capability. We propose Polynesia, a novel HTAP system that makes use of workload-optimized transactional and analytical islands to enable real-time analytics without sacrificing throughput. In Polynesia, transactional islands make use of conventional transactional database engines running on multicore CPUs, while analytical islands make use of custom co-designed algorithms and hardware that are placed inside memory. Our analytical islands are designed to alleviate the data movement and workload interference costs incurred in state-of-the-art HTAP systems, while still ensuring that data replicas for analytics workloads are kept up-to-date with the most recent version of the transactional data replicas. Polynesia outperforms three state-of-the-art HTAP systems (with a 1.7x/3.7x higher transactional/analytical throughput on average), while consuming less energy (48% lower than the best).

We conclude that the mechanisms proposed by this dissertation provide promising solutions to make PIM more effective and practical in computing systems.

8.2. Future Directions

The ideas and approaches proposed in this dissertation can potentially enables several future research directions. In this section, we discuss various directions that can be taken to address other challenges of PIM adoption in contemporary systems.

8.2.1. Extending Our Google Workload Analysis to Other Key Consumer Workloads

In Chapter 4, we analyze several widely-used Google consumer workloads to examine the suitability of PIM across those key workloads. In our analysis, we focus on applications that run on CPU cores in consumer devices. However, there are other important consumer applications that do not entirely run on CPU cores. One example is navigation apps such as Google Maps. Another popular example is 3D game applications. Both of these examples are some of the most widely-used consumer applications, and they both heavily rely on the GPU to render the scene and paints the pixels onto the screen. Camera-related applications are another important example of applications that do not entirely run on CPU cores. Emerging applications, such as 4K video streaming and recording [391], virtual reality (VR) [228], and augmented reality (AR) [17, 55, 146, 247], increasingly rely on the camera system (i.e., camera sensor, image sensor processing unit) to execute tasks. Thus, one future research direction is to extend our analysis to those workloads that rely on the GPU and the camera system and examine the benefit of PIM for those key workloads.

8.2.2. Extending CoNDA to Non-NDA Systems

Modern systems increasingly employ specialized accelerators to improve the energy efficiency of computing systems. As we discussed in Chapter 5, despite the significant benefits of accelerators, system challenges remain one of the main stumbling blocks to the mainstream adoption of specialized accelerators. These challenges can be largely mitigated by making the accelerator coherent with the rest of the system.

Despite the large body of prior works on specialized accelerators, few works attempt to address the coherence challenge. For example, FUSION [222] employs MESI coherence between on-chip accelerators and the CPU. However, when a system is equipped with many accelerators (which is the case for future systems), this fine-grained protocol generates a large number of unnecessary coherence messages between CPUs and accelerators and could potentially eliminate the majority of the benefits of employing specialized accelerators. Some other prior works [20, 204, 340] attempt to mitigate coherence overhead between accelerators such as GPU cores. While these proposals can be potentially applied to coherence problem between CPUs and on-chip accelerators, they are not readily compatible with existing coherence protocols, and require us to implement the new protocol across the entire system [340], which is a barrier to adoption.

One key advantage of CoNDA is that it significantly reduces the unnecessary coherence traffic using optimistic execution. Another key advantage of CoNDA is that it does not require the modification of existing coherence protocols elsewhere in the system. As a result, given these properties, we believe that the core idea of CoNDA can be extended to address coherence challenge between CPUs and on-chip accelerators. Note that CoNDA may not be directly applicable to on-chip accelerators coherence problem, as CoNDA is designed for CPU–NDA coherence and it is tailored toward the characteristics of NDA kernels (e.g., poor locality, low compute-intensity) as well as near-data accelerator features (e.g., simple fixed-function accelerators or small programmable cores that lack sophisticated ILP techniques). However, we believe that an architectural investigation can be done to see how we can extend the core idea of CoNDA to on-chip accelerators.

8.2.3. Exploiting Optimistic Execution in CoNDA for Efficient Scheduling of PIM Kernels and CPU Threads

As we discussed in Chapter 5, CoNDA uses optimistic execution to observe memory accesses and to gain insight into what part of the data will be accessed, and this execution model is key to eliminating coherence requests that are unnecessary. Optimistic execution can also be used to enable further optimizations for NDA-based systems, such as if a kernel should execute on an NDA or on the CPU. For example, CoNDA uses signature information during re-execution to further minimize data movement. CoNDA can use signature information to gain insight about the dirty cache lines in the processor LLC that will be accessed by PIM kernel during re-execution. Once conflicts happen, CoNDA can exploit this information to measure the fraction of the working set that is sitting in the processor LLC and needs to be written back to main memory. If the majority of the data accessed by the PIM kernel resides in the processor LLC, then offloading the PIM kernel to the near-data accelerator can generate more data movement due to the coherence traffic (i.e., writing back dirty data). In such a case, we let the processor thread executes the NDA kernel.

Another example is that CoNDA can use signature information to find whether both the PIM kernel and the processor threads access the same PIM data region concurrently. Those concurrent accesses, in which at least one of them updates the data, can lead to significant re-execution and unnecessary data movement for the near-data accelerator. In such a case, CoNDA can serialize accesses from the NDP kernel and the processor threads, as they are both accessing the same PIM data concurrently. To do so, CoNDA can use signature information to retrieve the set of addresses the PIM kernel touches during re-execution. It then re-executes the NDP kernel and prevents the processor threads from accessing those addresses during NDP kernel re-execution.

8.2.4. Extending Polynesia to Support Non-Relational Analytical Execution Engine

In Chapter 7, we focus on designing an instance of Polynesia that supports relational (SQL) transactional and analytical workloads. However, the term analytics is no longer limited to SQL analytics. Data is now produced in various formats, such as structured (e.g., records in DBMSs), semi-structured (e.g., JSON, XML), unstructured (e.g., text), graphs (e.g., social and interaction data), images, and videos. The diversity in data formats has given rise to many different types of analytics workloads (e.g., SQL/relational, graph processing, machine learning). Many emerging applications require *multiple* analytics workloads to run on the same data [29, 82, 200, 245, 304]. For example, SQL and machine learning analytics are used together to enable diabetes prediction [1,387]. Another example is GRfusion [156], which attempts to perform graph processing and SQL analytics on top of relational data. A modern HTAP system needs to support both high update rates as well as the ability to run diverse analytics on the data (e.g., [60, 215, 296]). One potential future research direction is to examine how we can use Polynesia framework to support various types of analytics workloads and what software/hardware co-design techniques we need to efficiently support real-time analysis for these emergying HTAP systems.

8.2.5. Automating the Cluster Identification in Mensa

As we discussed in Chapter 6, Mensa's scheduler relies on two pieces of information to generate a mapping between layers and accelerators: (1) the characteristics of each cluster; and (2) which hardware accelerator is best suited for each cluster. In our work, we assume that, similar to how chipset drivers are configured, this information is generated once during the initial setup of a system and then handed over to the scheduler. However, generating the first piece (i.e., cluster identification) could be challenging as it involves a comprehensive analysis across all layers from all models. This is especially challenging (and may lead to sub-optimal cluster formation) because NN algorithms are evolving rapidly, which has led to a myriad of NN models. As a result, one future research direction is how we can automate the process of identifying clusters across a wide range of models. One potential solution to address this challenge is to employ automated Neural Architecture Search (NAS) [140] methods to identify clusters. NAS methods have recently gained popularity in the deep learning community to design models, as opposed to conventional approach of hand tuning model hyperparameters.

8.2.6. Addressing Security Challenges of PIM

While PIM can help us mitigate data movement costs, it can also expose computing systems to security vulnerabilities. For example, one potential security issue is memory corruption [26]. Many proposed PIM architectures [15, 43, 44, 168, 321] add new instructions to existing ISAs to execute PIM kernel computations. At the same time, many PIM architectures [14,43,44,93] assume that PIM kernels operate only on physical addresses to avoid the overhead of maintaining address translation. These systems often do not have fine-grain per-page protections [34]. Without proper protection for these extended ISA, an attacker can write a malicious PIM kernel which uses these extended ISA to access out-of-the-bound memory addresses and to corrupt memory locations (e.g., tampering with a DNN model parameters or activations [244]). To make the matter worse, PIM kernels often operate on a large amount of data and perform bulk operations, exacerbating the memory corruption issue. Another example of potential security issues is cache side-channel attacks [26, 84]. If the system maintains coherence between PIM and the CPUs, a malicious PIM kernel can potentially generate/monitor coherence requests to the LLC and capture timing characteristics of LLC memory accesses, allowing the attacker to recover information related to computation on the CPU side (e.g. information related to a DNN model running on the CPU side, encryption keys). Thus, one important future research direction is to identify the security challenges of PIM and propose system and architectural solutions to address them.

Appendix A

Other Works by the Author

In addition to the works presented in this thesis, I have also contributed to several other research works done in collaboration with fellow graduate students at CMU and ETH. In this section, I briefly discuss these works.

Gather-Scatter DRAM [322] is a low-cost substrate that enables the memory controller to efficiently gather or scatter data with different non-unit strided access patterns. Our mechanism exploits the fact that multiple DRAM chips contribute to each cache line access. GS-DRAM maps values accessed by different strided patterns to different chips, and uses a per-chip column translation logic to access data with different patterns using significantly fewer memory accesses than existing DRAM interfaces. Our framework requires no changes to commodity DRAM chips, and very few changes to the DRAM module, the memory interface, and the processor architecture.

IMPICA [168] is an in-memory accelerator for performing pointer chasing operations in 3D-stacked memory. We identify two major challenges in the design of such an in-memory accelerator: (1) the parallelism challenge and (2) the address translation challenge. We provide new solutions to these two challenges: (1) address-access decoupling solves the parallelism challenge by decoupling the address generation from memory accesses in pointer chasing operations and exploiting the idle time during memory accesses to execute multiple pointer chasing operations in parallel, and (2) the region-based page table in 3D-stacked memory solves the address translation challenge by tracking only those limited set of virtual memory regions that are accessed by pointer chasing operations.

Ambit [321] is a new accelerator that performs bulk bitwise operations within a DRAM chip by exploiting the analog operation of DRAM. Ambit consists of two components. The first component uses simultaneous activation of three DRAM rows to perform bulk bitwise AND/OR operations. The second component uses the inverters present in each sense amplifier to perform bulk bitwise NOT operations. With these two components, Ambit can perform any bulk bitwise operation efficiently in DRAM. Ambit is generally applicable to any memory device that uses DRAM (e.g, 3D-stacked DRAM, embedded DRAM).

GenASM [48] is an approximate string matching (ASM) acceleration framework for genome sequence analysis built upon a modified and enhanced Bitap algorithm [32]. GenASM performs bitvector-based ASM, which can accelerate multiple steps of genome sequence analysis. We co-design our highly parallel, scalable and memory-efficient algorithms with low-power and area-efficient hardware accelerators. We show that GenASM can accelerate several different use cases of ASM in genome sequence analysis for both short and long reads: read alignment, pre-alignment filtering, and edit distance calculation.

In collaboration with Geraldo Oliveira, we comprehensively evaluate the performance implications of leveraging novel 3D-based NVM devices in consumer devices [284]. We compare the average and 99th-percentile latencies of a system equipped with a 16 GB Intel Optane SSD as the primary swap device to a baseline system with twice the DRAM size. We observe that by enabling an NVM-based swap space, we can obtain performance benefits for an interactive workload (Google Chrome). To reduce the impact of I/O activity in the system, we propose several system optimizations based on key characteristics of our workload.

Appendix B

Bibliography

[1] IBM Watson To Power Medtronic's Diabetes App.

[2] The Transaction Processing Council. TPC-C Benchmark, 2007.

[3] The Transaction Processing Council. TPC-H Benchmark, 2007.

[4] Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation, 2013.

[5] Oracle GoldenGate 12c: Real-Time Access to Real-Time Information, 2015.

[6] Cisco Global Cloud Index: Forecast and Methodology, 2016-2021, 2016.

[7] HarperDB, 2017.

[8] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 466–475. IEEE, 2007.

[9] D. Abts. Lost in the Bermuda Triangle: Complexity, Energy, and Performance. In *WCED*, 2006.

[10] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors. *TPDS*, 2005.

[11] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks. Fathom: Reference Workloads for Modern Deep Learning Methods. In *IISWC*, 2016.

[12] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *HPCA*, 2017.

[13] A. Agrawal, G. H. Loh, and J. Tuck. Leveraging Near Data Processing for High-Performance Checkpoint/Restart. In *SC*, 2017.

[14] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015.

[15] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *ISCA*, 2015.

[16] B. Akin, F. Franchetti, and J. C. Hoe. Data Reorganization in Memory Using 3D-Stacked DRAM. In *ISCA*, 2015.

[17] A. Al-Shuwaili and O. Simeone. Energy-Efficient Resource Allocation for Mobile Edge Computing-Based Augmented Reality Applications. *IEEE Wireless Communications Letters*, 2017.

[18] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ISCA*, 2016.

[19] Alexa Internet, Inc. Website Traffic, Statistics and Analytics. `http://www.alexa.com/siteinfo/`.

[20] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood. Lazy Release Consistency for GPUs. In *MICRO*, 2016.

[21] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *MICRO*, 2016.

[22] M. Alzantot, Y. Wang, Z. Ren, and M. B. Srivastava. RSTensorFlow: GPU Enabled TensorFlow for Deep Learning on Commodity Android Devices. In *EMDL*, 2017.

[23] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA*, 2005.

[24] S. Angizi, Z. He, A. S. Rakin, and D. Fan. Cmp-pim: An energy-efficient comparator-based processing-in-memory neural network accelerator. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[25] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki. The Case for Heterogeneous HTAP. In *CIDR*, 2017.

[26] M. T. Arafin and Z. Lu. Security challenges of processing-in-memory systems. In *GVLSI*, 2020.

[27] ARM Holdings PLC. ARM Cortex-A7. `https://developer.arm.com/ip-products/processors/cortex-a/cortex-a7`.

[28] ARM Holdings PLC. ARM Cortex-R8. `https://developer.arm.com/products/processors/cortex-r/cortex-r8`.

[29] V. Arora, F. Nawab, D. Agrawal, and A. E. Abbadi. Multi-representation Based Data Processing Architecture for IoT Applications. In *ICDCS*, 2017.

[30] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*, 2016.

[31] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *MICRO*, 2016.

[32] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Communications of The ACM*, 1992.

[33] R. Barber, C. Garcia-Arellano, R. Grosman, G. Lohman, C. Mohan, R. Muller, H. Pirahesh, V. Raman, R. Sidle, A. Storm, Y. Tian, P. Tozun, and Y. Wu. WiSer: A Highly Available HTAP DBMS for IoT Applications. arxiv:1908.01908 [cs.DB], 2019.

[34] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient Virtual Memory for Big Memory Servers. In *ISCA*, 2013.

[35] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky. Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[36] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.*, 1995.

[37] P. A. Bernstein and N. Goodman. Multiversion concurrency control&mdash;theory and algorithms. *ACM Trans. Database Syst.*, 1983.

[38] N. Binkert, B. Beckman, A. Saidi, G. Black, and A. Basu. The gem5 Simulator. *Comp. Arch. News*, 2011.

[39] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, 2009.

[40] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 1970.

[41] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. `https://csde.washington.edu/~mbw/OLD/UNIX/zfs_lite.pdf`, 2007.

[42] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *ASPLOS*, 2018.

[43] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu. Conda: Efficient cache coherence support for near-data accelerators. In *ISCA*, 2019.

[44] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *IEEE CAL*, 2017.

[45] F. Bossen, B. Bross, K. Suhring, and D. Flynn. HEVC Complexity and Implementation Analysis. *IEEE CSVT*, 2012.

[46] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *WWW*, 1998.

[47] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable Isolation for Snapshot Databases. In *SIGMOD*, 2008.

[48] D. S. Cali, G. S. Kalsi, Z. Bingl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Norion, A. Scibisz, S. Subramoneyon, C. Alkan, S. Ghose, and O. Mutlu. Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *MICRO*, 2020.

[49] Q. Cao, N. Balasubramanian, and A. Balasubramanian. MobiRNN: Efficient Recurrent Neural Network Execution on Mobile GPU. In *EMDL*, 2017.

[50] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX ATC*, 2010.

[51] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, 2007.

[52] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, 2006.

[53] G. Chadha, S. Mahlke, and S. Narayanasamy. EFetch: Optimizing Instruction Fetch for Event-Driven Web Applications. In *PACT*, 2014.

[54] G. Chadha, S. Mahlke, and S. Narayanasamy. Accelerating Asynchronous Programs Through Event Sneak Peek. In *ISCA*, 2015.

[55] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui. Mobile Augmented Reality Survey: From Where We Are to Where We Go. *IEEE Access*, 2017.

[56] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ASPLOS*, 2014.

[57] Y. Chen, T. Yang, J. Emer, and V. Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *JETCAS*, 2019.

[58] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *JSSC*, 2017.

[59] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *ISCA*, 2016.

[60] F. Chirigati, J. Simon, M. Hirzel, and J. Freire. Virtual lightweight snapshots for consistent analytics in NoSQL stores. In *ICDE*, 2016.

[61] J.-A. Choi and Y.-S. Ho. Deblocking Filter Algorithm with Low Complexity for H.264 Video Coding. In *PCM*, 2008.

[62] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc. A roofline model of energy. In *IPDPS*, 2013.

[63] C. Chou, P. Nair, and M. K. Qureshi. Reducing Refresh Power in Mobile Devices with Morphable ECC. In *DSN*, 2015.

[64] Chromium Project. Blink Rendering Engine. `https://www.chromium.org/blink`.

[65] Chromium Project. Catapult: Telemetry. `https://chromium.googlesource.com/catapult/+/HEAD/telemetry/README.md`.

[66] Chromium Project. GPU Rasterization in Chromium. `https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome`, 2014.

[67] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS*, 2014.

[68] Cisco Systems, Inc. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html`, 2017.

[69] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *MobiSys*, 2010.

[70] H. Deng, X. Zhu, and Z. Chen. An Efficient Implementation for H.264 Decoder. In *ICCSIT*, 2010.

[71] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang. Dracc: a dram based accelerator for accurate cnn inference. In *DAC*, 2018.

[72] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[73] T. M. T. Do, J. Blom, and D. Gatica-Perez. Smartphone Usage in the Wild: A Large-Scale Analysis of Applications and Context. In *ICMI*, 2011.

[74] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *CVPR*, 2015.

[75] DRAMSim2. http://www.eng.umd.edu/ blj/dramsim/.

[76] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca. The Architecture of the DIVA Processing-in-memory Chip. In *ICS*, 2002.

[77] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. The Mondrian Data Engine. In *ISCA*, 2017.

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

[79] P. Dubroy and R. Balakrishnan. A Study of Tabbed Browsing Among Mozilla Firefox Users. In *CHI*, 2010.

[80] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *ISCA*, 2018.

[81] D. G. Elliott, W. M. Snelgrove, and M. Stumm. Computational ram: A memory-simd hybrid and its application to dsp. In *1992 Proceedings of the IEEE Custom Integrated Circuits Conference*, 1992.

[82] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik. A demonstration of the bigdawg polystore system. *Proc. VLDB Endow.*, 2015.

[83] eMarketer, Inc. Slowing Growth Ahead for Worldwide Internet Audience. `https://www.emarketer.com/article/slowing-growth-ahead-worldwide-internet-audience/1014045?soc1001`, 2016.

[84] S. S. Ensan, K. Nagarajan, M. N. I. Khan, and S. Ghosh. Scare: Side channel attack on in-memory computing for reverse engineering, 2020.

[85] Ericsson, Inc. Ericsson Mobility Report: On the Pulse of the Networked Society. `https://www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf`, 2015.

[86] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, 2011.

[87] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 1976.

[88] Facebook, Inc. Instagram. `https://www.instagram.com/`.

[89] B. Falsafi, M. Stan, K. Skadron, N. Jayasena, Y. Chen, J. Tao, R. Nair, J. Moreno, N. Muralimanohar, K. Sankaralingam, and C. Estan. Near-Memory Data Services. *IEEE Micro*, 2016.

[90] W. Fang, B. J. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 2010.

[91] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien. UDP: A Programmable Accelerator for Extract-Transform-Load Workloads and More. In *MICRO*, 2017.

[92] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database – an architecture overview. *IEEE Data Eng. Bull.*, 2012.

[93] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *HPCA*, 2015.

[94] I. Fernandez, R. Quislant, C. Gianoula, M. Alser, J. Gmez-Luna, E. Gutirrez, O. Plata, and O. Mutlu. Natsa: A near-data processing accelerator for time series analysis. In *ICCD*, 2020.

[95] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, 2018.

[96] F. Gao, G. Tziantzioulis, and D. Wentzlaff. Computedram: In-memory compute using off-the-shelf drams. In *MICRO*, 2019.

[97] M. Gao, G. Ayers, and C. Kozyrakis. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *PACT*, 2015.

[98] M. Gao and C. Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *HPCA*, 2016.

[99] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *ASPLOS*, 2017.

[100] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *ASPLOS*, 2019.

[101] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, X. Wen, R. Ren, C. Zheng, X. He, H. Ye, H. Tang, Z. Cao, S. Zhang, and J. Dai. Bigdatabench: A scalable and unified big data and ai benchmark suite, 2018.

[102] S. Ghose, A. Boroumand, J. S. Kim, J. Gmez-Luna, and O. Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 2019.

[103] S. Ghose, A. Boroumand, J. S. Kim, J. Gmez-Luna, and O. Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 2019.

[104] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu. Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. arxiv:1802.00320 [cs.AR], 2018.

[105] S. Ghose, T. Li, N. Hajinazar, D. Senol Cali, and O. Mutlu. Demystifying Complex Workload–DRAM Interactions: An Experimental Study. In *SIGMETRICS*, 2019.

[106] J. Giceva and M. Sadoghi. *Hybrid OLTP and OLAP*. 2018.

[107] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: An architecture for scale-out of olxp workloads. *Proc. VLDB Endow.*, 2015.

[108] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch. HARE: Hardware Accelerator for Regular Expressions. In *MICRO*, 2016.

[109] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 1995.

[110] J. R. Goodman. Using Cache Memory to Reduce Processor-memory Traffic. In *ISCA*, 1983.

[111] Google LLC. Android. `https://www.android.com/`.

[112] Google LLC. Chrome Browser. `https://www.google.com/chrome/browser/`.

[113] Google LLC. Chromebook. `https://www.google.com/chromebook/`.

[114] Google LLC. Edge TPU.

[115] Google LLC. Edge TPU Compiler. `https://coral.ai/docs/edgetpu/compiler/`.

[116] Google LLC. gemmlowp: a small self-contained low-precision GEMM library. `https://github.com/google/gemmlowp`.

[117] Google LLC. Gmail. `https://www.google.com/gmail/`.

[118] Google LLC. Google Calendar. `https://calendar.google.com/`.

[119] Google LLC. Google Docs. `https://docs.google.com/`.

[120] Google LLC. Google Hangouts. `https://hangouts.google.com/`.

[121] Google LLC. Google Photos. `https://photos.google.com/`.

[122] Google LLC. Google Search. `https://www.google.com/`.

[123] Google LLC. Google Search: About Google App. `https://www.google.com/search/about/`.

[124] Google LLC. Google Translate. `https://translate.google.com/`.

[125] Google LLC. Google Translate App. `https://translate.google.com/intl/en/about/`.

[126] Google LLC. Skia Graphics Library. `https://skia.org/`.

[127] Google LLC. TensorFlow Lite. `https://www.tensorflow.org/lite/`.

[128] Google LLC. TensorFlow: Mobile. `https://www.tensorflow.org/mobile/`.

[129] Google LLC. TensorFlow Models on the Edge TPU. `https://coral.ai/docs/edgetpu/models-intro/`.

[130] Google LLC. YouTube. `https://www.youtube.com/`.

[131] Google LLC. YouTube for Press. `https://www.youtube.com/yt/about/press/`.

[132] D. Gope, D. J. Schlais, and M. H. Lipasti. Architectural Support for Server-Side PHP Processing. In *ISCA*, 2017.

[133] G. Graefe. Encapsulation of parallelism in the volcano query processing system. *SIGMOD*, 1990.

[134] A. Grange, P. de Rivaz, and J. Hunt. VP9 Bitstream & Decoding Process Specification. `http://storage.googleapis.com/downloads.webmproject.org/docs/vp9/vp9-bitstream-specification-v0.6-20160331-draft.pdf`.

[135] A. Graves. Sequence transduction with recurrent neural networks. *arXiv*, 2012.

[136] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: A main memory hybrid storage engine. *Proc. VLDB Endow.*, 2010.

[137] B. Gu, A. S. Yoon, D. Bae, I. Jo, J. Lee, J. Yoon, J. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. Biscuit: A framework for near-data processing of big data workloads. In *ISCA*, 2016.

[138] S. Gudaparthi, S. Narayanan, R. Balasubramonian, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon. Wire-aware architecture and dataflow for cnn accelerators. In *MICRO*, 2019.

[139] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti. 3D-Stacked Memory-Side Acceleration: Accelerator and System Design. In *WoNDP*, 2014.

[140] S. Gupta and B. Akin. Accelerator-aware neural network design using automl, 2020.

[141] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G. Wei, and D. Brooks. Masr: A modular accelerator for sparse rnns. In *PACT*, 2019.

[142] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-System Analysis and Characterization of Interactive Smartphone Applications. In *IISWC*, 2011.

[143] H. Habli, J. Lilius, and J. Ersfolk. Analysis of Memory Access Optimization for Motion Compensation Frames in MPEG-4. In *SOC*, 2009.

[144] R. Hadidi, L. Nai, H. Kim, and H. Kim. CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-in-Memory. *ACM TACO*, 2017.

[145] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky. Imaging: In-memory algorithms for image processing. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2018.

[146] M. Halpern, Y. Zhu, and V. J. Reddi. Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction. In *HPCA*, 2016.

[147] T. J. Ham, J. L. Aragón, and M. Martonosi. DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures. In *MICRO*, 2015.

[148] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *MICRO*, 2016.

[149] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *ASPLOS*, 1998.

[150] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, 2004.

[151] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and B. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*, 2016.

[152] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt. Accelerating dependent cache misses with an enhanced memory controller. In *ISCA*, 2016.

[153] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *ISCA*, 2016.

[154] M. Hashemi, O. Mutlu, and Y. N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *MICRO*, 2016.

[155] M. Hashemi, O. Mutlu, and Y. N. Patt. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *MICRO*, 2016.

[156] M. S. Hassan, T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi. Empowering in-memory relational database engines with native graph processing. *CoRR*, 2017.

[157] K. He, X. Zhang, S. Ren, and J. Sun. Identity Mappings in Deep Residual Networks. In *ECCV*, 2016.

[158] Y. He, T. N. Sainath, R. Prabhavalkar, I. McGraw, R. Alvarez, D. Zhao, D. Rybach, A. Kannan, Y. Wu, R. Pang, Q. Liang, D. Bhatia, Y. Shangguan, B. Li, G. Pundak, K. C. Sim, T. Bagby, S. Chang, K. Rao, and A. Gruenstein. Streaming end-to-end speech recognition for mobile devices. In *ICASSP*, 2019.

[159] B. Heater. As Chromebook Sales Soar in Schools, Apple and Microsoft Fight Back. `https://techcrunch.com/2017/04/27/as-chromebook-sales-soar-in-schools-apple-and-microsoft-fight-back/`, 2017.

[160] J. Hennessy, M. Heinrich, and A. Gupta. Cache-Coherent Distributed Shared Memory: Perspectives on Its Development and Future Challenges. *Proc. IEEE*, 1999.

[161] M. Herlihy, J. Eliot, and B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, 1993.

[162] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. *ISCA*, 1993.

[163] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim. Accelerating Linked-List Traversal Through Near-Data Processing. In *PACT*, 2016.

[164] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro. H.264/AVC Baseline Profile Decoder Complexity Analysis. *CSVT*, 2003.

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

[166] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[167] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Conner, N. Vijaykumar, O. Mutlu, and S. Keckler. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *ISCA*, 2016.

[168] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016.

[169] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016.

[170] HTTP Archive. `http://httparchive.org/`.

[171] Y. Huang, Z. Zha, M. Chen, and L. Zhang. Moby: A Mobile Benchmark Suite for Architectural Simulators. In *ISPASS*, 2014.

[172] D. A. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proc. IRE*, 1952.

[173] D. Hwang. Native One-Copy Texture Uploads. `https://01.org/chromium/2016/native-one-copy-texture-uploads-for-chrome-OS`, 2016.

[174] Hybrid Memory Cube Consortium. HMC Specification 2.0, 2014.

[175] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

[176] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(1):40–45, 2012.

[177] Intel Corp. Intel Celeron Processor N3060. `https://ark.intel.com/products/91832/Intel-Celeron-Processor-N3060-2M-Cache-up-to-2_48-GHz`.

[178] Intel Corp. Intel Chipset 89xx Series. `http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/scaling-acceleration-capacity-brief.pdf`.

[179] Intel Corp. Intel Movidius Neural Compute Stick.

[180] Intel Corp. Software vs. GPU Rasterization in Chromium. `https://software.intel.com/en-us/articles/software-vs-gpu-rasterization-in-chromium`.

[181] M. H. Ionica and D. Gregg. The movidius myriad architecture's potential for scientific computing. *IEEE Micro*, 2015.

[182] Z. Istvan, L. Woods, and G. Alonso. Histograms As a Side Effect of Data Movement for Big Data. In *SIGMOD*, 2014.

[183] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan. Computing in memory with spin-transfer torque magnetic ram. *TVLSI*, 2018.

[184] J. Jeddeloh and B. Keeth. Hybrid Memory Cube New DRAM Architecture Increases Density and Performance. In *VLSIT*, 2012.

[185] JEDEC Solid State Technology Assn. JESD79-3F: DDR3 SDRAM Standard, 2012.

[186] JEDEC Solid State Technology Assn. JESD235: High Bandwidth Memory (HBM) DRAM, 2013.

[187] JEDEC Solid State Technology Assn. JESD79-4B: DDR4 SDRAM Standard, 2017.

[188] JEDEC Solid State Technology Assn. JESD209-4C: Low Power Double Data Rate 4 (LPDDR4) Standard, January 2020.

[189] JEDEC Solid State Technology Assn., JESD229 . Wide I/O Single Data Rate (Wide I/O SDR) Standard,, 2011.

[190] S. Jennings. Transparent Memory Compression in Linux. `https://events.static.linuxfound.org/sites/events/files/slides/tmc_sjennings_linuxcon2013.pdf`, 2013.

[191] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[192] E. Kalali and I. Hamzaoglu. A Low Energy HEVC Sub-Pixel Interpolation Hardware. In *ICIP*, 2014.

[193] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[194] J. Kane and Q. Yang. Compression Speed Enhancements to LZO for Multi-Core Systems. In *SBAC-PAD*, 2012.

[195] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ISCA*, 2015.

[196] S. Kanev, S. L. Xi, G.-Y. Wei, and D. Brooks. Mallacc: Accelerating Memory Allocation. In *ASPLOS*, 2017.

[197] U. Kang, H. soo Yu, C. Park, H. Zheng, J. B. Halbert, K. S. Bains, S.-J. Jang, and J. S. Choi. Co-architecting controllers and dram to enhance dram process scaling. 2014.

[198] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ASPLOS*, 2017.

[199] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *ICCD*, 2012.

[200] K. Kara, K. Eguro, C. Zhang, and G. Alonso. ColumnML: column-store machine learning with on-the-fly data transformation. *Proceedings of the VLDB Endowment*, 2018.

[201] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C. Wu, M. Hempstead, and X. Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *ISCA*, 2020.

[202] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 2011.

[203] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 2011.

[204] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel. WAYPOINT: Scaling Coherence to Thousand-Core Architectures. In *PACT*, 2010.

[205] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, 2011.

[206] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay. NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *ISCA*, 2016.

[207] G. Kim, N. Chatterjee, M. O'Connor, and K. Hsieh. Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs. In *SC*, 2017.

[208] G. Kim, N. Chatterjee, M. O'Connor, and K. Hsieh. Toward standardized near-data processing with unrestricted data placement for gpus. In *SC*, 2017.

[209] G. Kim, J. Kim, J. H. Ahn, and J. Kim. Memory-Centric System Interconnect Design with Hybrid Memory Cubes. In *PACT*, 2013.

[210] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu. GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies. *BMC Genomics*, 2018.

[211] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ISCA*, 2014.

[212] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.

[213] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.

[214] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE CAL*, 2015.

[215] H. Kimura, A. Simitsis, and K. J. Wilkinson. Janus: Transaction Processing of Navigation and Analytic Graph Queries on Many-Core Servers. In *CIDR*, 2017.

[216] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.*, 2015.

[217] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-Memory Databases. In *MICRO*, 2013.

[218] P. M. Kogge. EXECUBE: A New Architecture for Scaleable MPPs. In *ICPP*, 1994.

[219] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *ICS*, 1998.

[220] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *CACM*, 2017.

[221] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core cpus. *Proc. VLDB Endow.*, 2011.

[222] S. Kumar, A. Shriraman, and N. Vedula. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. In *ISCA*, 2015.

[223] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *TODB*, 1981.

[224] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *MICRO*, 2019.

[225] H. Kwon, A. Samajdar, and T. Krishna. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *ASPLOS*, 2018.

[226] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. H. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. Oracle Database In-Memory: A Dual Format In-Memory Database. In *ICDE*, 2015.

[227] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, M. S, V. Raja, M. Roth, E. Soylemez, and M. Zat. Oracle database in-memory: A dual format in-memory database. 2015.

[228] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai. Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices. In *MobiCom*, 2017.

[229] M. J. Langroodi, J. Peters, and S. Shirmohammadi. Decoder-Complexity-Aware Encoding of Motion Compensation for Multiple Heterogeneous Receivers. *TOMM*, 2015.

[230] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos. Real-time analytical processing with sql server. *PVLDB*, 2015.

[231] P.-A. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos. Real-time Analytical Processing with SQL Server. *Proc. VLDB Endow.*, 2015.

[232] C. Lee and Y. Yu. Design of a Motion Compensation Unit for H.264 Decoder Using 2-Dimensional Circular Register Files. In *ISOCC*, 2008.

[233] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu. Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. *ACM TACO*, 2016.

[234] J. Lee, S. Moon, K. H. Kim, D. H. Kim, S. K. Cha, W.-S. Han, C. G. Park, H. J. Na, and J.-Y. Lee. Parallel replication across formats in sap hana for scaling out mixed oltp/olap workloads. *PVLDB*, 2017.

[235] J. Lee, Y. Solihin, and J. Torrettas. Automatically Mapping Code on an Intelligent Memory Architecture. In *HPCA*, 2001.

[236] J. H. Lee, J. Sim, and H. Kim. Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models. In *PACT*, 2015.

[237] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*, 2014.

[238] C. Lemke, K.-U. Sattler, F. Faerber, and A. Zeier. Speeding up queries in column stores: A case for compression. In *DaWak*, 2010.

[239] P. Lewis. Avoiding Unnecessary Paints. `https://www.html5rocks.com/en/tutorials/speed/unnecessary-paints/`, 2013.

[240] J. Li, R. Zhao, H. Hu, and Y. Gong. Improving rnn transducer modeling for end-to-end speech recognition. In *ASRU*, 2019.

[241] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *MICRO*, 2017.

[242] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[243] T. Li, C. An, X. Xiao, A. T. Campbell, and X. Zhou. Real-Time Screen-Camera Communication Behind Any Scene. In *MobiSys*, 2015.

[244] W. Li, Y. Wang, H. Li, and X. Li. Leveraging memory pufs and pim-based encryption to secure edge deep learning systems. In *VTS*, 2019.

[245] H. Lim, Y. Han, and S. Babu. How to fit when no one size fits. In *CIDR*, 2013.

[246] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *ISCA*, 2013.

[247] F. Liu, P. Shu, H. Jin, L. Ding, J. Yu, D. Niu, and B. Li. Gearing Resource-Poor Mobile Devices with Powerful Clouds: Architectures, Challenges, and Applications. *IEEE Wireless Communications*, 2013.

[248] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao. Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach. In *MICRO*, 2018.

[249] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu. Concurrent data structures for near-memory computing. In *SPAA*, 2017.

[250] G. H. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. In *ISCA*, 2008.

[251] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *HPCA*, 2017.

[252] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu. Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator. In *MICRO*, 2012.

[253] M. Mahmoud, B. Zheng, A. D. Lascorz, F. Heide, J. Assouline, P. Boucher, E. Onzon, and A. Moshovos. IDEAL: Image Denoising Accelerator. In *MICRO*, 2017.

[254] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *ISCA*, 2000.

[255] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD*, 2017.

[256] H. Mao. Hardware Acceleration for Memory to Memory Copies. Master's thesis, 2017.

[257] MemSQL, Inc. MemSQL. `http://www.memsql.com/`.

[258] Mentor Graphics Corp. Catapult High-Level Synthesis. `https://www.mentor.com/hls-lp/catapult-high-level-synthesis/`.

[259] Microsoft Corp. Skype. `https://www.skype.com/`.

[260] A. Mirhosseini, A. Agrawal, and J. Torrellas. Survive: Pointer-Based In-DRAM Incremental Checkpointing for Low-Cost Data Persistence and Rollback-Recovery. *IEEE CAL*, 2017.

[261] N. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot. Sort vs. Hash Join Revisited for Near-Memory Execution. In *ASBD*, 2007.

[262] B. Moatamed, Arjun, F. Shahmohammadi, R. Ramezani, A. Naeim, and M. Sarrafzadeh. Low-Cost Indoor Health Monitoring System. In *BSN*, 2016.

[263] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *HPCA*, 2006.

[264] A. Mosenia, S. Sur-Kolay, A. Raghunathan, and N. K. Jha. CABA: Continuous Authentication Based on BioAura. *IEEE TC*, 2017.

[265] A. Mosenia, S. Sur-Kolay, A. Raghunathan, and N. K. Jha. Wearable Medical Sensor-Based System Design: A Survey. *MSCS*, 2017.

[266] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. Scyper: A hybrid oltp and olap distributed main memory database system for scalable real-time analytics. In *BTW*, 2013.

[267] K. Murakami, S. Shirakawa, and H. Miyajima. Parallel processing ram chip with 256 mb dram and quad processors. In *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers*, 1997.

[268] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *MICRO*, 2011.

[269] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning. In *MICRO*, 2011.

[270] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *MICRO*, 2007.

[271] O. Mutlu, S. Ghose, J. Gomez-Luna, and R. Ausavarungnirun. Processing Data Where It Makes Sense: Enabling In-Memory Computation. *MICPRO*, 2019.

[272] O. Mutlu, S. Ghose, J. Gmez-Luna, and R. Ausavarungnirun. A modern primer on processing in memory, 2020.

[273] O. Mutlu and J. S. Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[274] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das. VIP: Virtualizing IP Chains on Handheld Platforms. In *ISCA*, 2015.

[275] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *HPCA*, 2017.

[276] G. Narancic, P. Judd, D. Wu, I. Atta, M. Elnacouzi, J. Zebchuk, J. Albericio, N. E. Jerger, A. Moshovos, K. Kutulakos, and S. Gadelrab. Evaluating the Memory System Behavior of Smartphone Workloads. In *SAMOS*, 2014.

[277] Net Applications. Market Share Statistics for Internet Technologies. `https://www.netmarketshare.com/`.

[278] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 2011.

[279] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, 2015.

[280] A. M. Nia, M. Mozaffari-Kermani, S. Sur-Kolay, A. Raghunathan, and N. K. Jha. Energy-Efficient Long-term Continuous Personal Health Monitoring. *MSCS*, 2015.

[281] Nielsen Norman Group. Page Parking: Millennials' Multi-Tab Mania. `https://www.nngroup.com/articles/multi-tab-page-parking/`.

[282] NVIDIA Corp. NVIDIA Jetson Nano.

[283] M. F. X. J. Oberhumer. LZO Real-Time Data Compression Library. `http://www.oberhumer.com/opensource/lzo/`, 2018.

[284] G. Oliveira, S. Ghose, J. Gmez-Luna, A. Boroumand, A. Savery, S. Rao, S. Qazi, G. Grignou, R. Thakur, E. Shiu, and O. Mutlu. Understanding the impact of emerging non-volatile memory in off-the-shelf consumer devices. In *SIGMETRICS*, 2021.

[285] L. E. Olson, M. D. Hill, and D. A. Wood. Crossing Guard: Mediating Host-Accelerator Coherence Interactions. In *ASPLOS*, 2017.

[286] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood. Synchronization using remote-scope promotion. In *ASPLOS*, 2015.

[287] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *ISCA*, 1998.

[288] F. Özcan, Y. Tian, and P. Tözün. Hybrid transactional/analytical processing: A survey. In *SIGMOD*, 2017.

[289] D. Pandiyan, S.-Y. Lee, and C.-J. Wu. Performance, Energy Characterizations and Architectural Implications of an Emerging Mobile Platform Benchmark Suite – MobileBench. In *IISWC*, 2013.

[290] D. Pandiyan and C.-J. Wu. Quantifying the Energy Cost of Data Movement for Emerging Smartphone Workloads on Mobile Platforms. In *IISWC*, 2014.

[291] M. S. Papamarcos and J. H. Patel. A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ISCA*, 1984.

[292] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *ISCA*, 2017.

[293] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 1997.

[294] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities. In *PACT*, 2016.

[295] J. Picorel, D. Jevdjic, and B. Falsafi. Near-Memory Address Translation. In *PACT*, 2017.

[296] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquín, and D. Kossmann. Fast scans on key-value stores. *Proc. VLDB Endow.*, 2017.

[297] B. Popper. Google Services Monthly Active Users. `https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users`, 2017.

[298] J. Power, A. Basu, J. Gu, M. Hill, and D. Wood. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 457–467. IEEE/ACM, 2013.

[299] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling up concurrent main-memory column-store scans: Towards adaptive numa-aware data and task placement. *Proc. VLDB Endow.*, 2015.

[300] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *VLDB Endowment*, 2015.

[301] S. H. Pugsley, A. Deb, R. Balasubramonian, and F. Li. Fixed-function hardware sorting accelerators for near data MapReduce execution. In *ICCD*, 2015.

[302] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. NDC: Analyzing the Impact of 3D-stacked Memory+Logic Devices on MapReduce Workloads. In *ISPASS*, 2014.

[303] Qualcomm Technologies, Inc. Snapdragon 835 Mobile Platform. `https://www.qualcomm.com/products/snapdragon/processors/835`.

[304] J. Ramnarayan, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav. Snappydata : Streaming , transactions , and interactive analytics in a unified engine. In *CIDR*, 2016.

[305] K. Rao, H. Sak, and R. Prabhavalkar. Exploring architectures, data and units for streaming end-to-end speech recognition with rnn-transducer, 2018.

[306] B. Reagen, R. Adolf, Y. S. Shao, G. Y. Wei, and D. Brooks. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *IISWC*, 2014.

[307] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *ISCA*, 2016.

[308] J. Ren and N. Kehtarnavaz. Comparison of Power Consumption for Motion Compensation and Deblocking Filters in High Definition Video Coding. In *ISCE*, 2007.

[309] P. V. Rengasamy, H. Zhang, N. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. Characterizing Diverse Handheld Apps for Customized Hardware Acceleration. In *IISWC*, 2017.

[310] S. H. S. Rezaei, M. Modarressi, R. Ausavarungnirun, M. Sadrosadati, O. Mutlu, and M. Daneshtalab. Nom: Network-on-memory for inter-bank data transfer in highly-banked memories. *IEEE CAL*, 2020.

[311] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree Filesystem. *ACM TOS*, 2013.

[312] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen. Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild. In *IMC*, 2015.

[313] F. Ross. Migrating to LPDDR3: An Overview of LPDDR3 Commands, Operations, and Functions. In *JEDEC LPDDR3 Symposium*, 2012.

[314] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim. L-store: A real-time oltp and olap system. In *EDBT*, 2018.

[315] H. Sak, A. W. Senior, and F. Beaufays. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. *arXiv*, 2014.

[316] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *MICRO*, 2007.

[317] SAP SE. SAP HANA. `http://www.hana.sap.com/`.

[318] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. The Dirty Block Index. In *ISCA*, 2014.

[319] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Fast Bulk Bitwise AND and OR in DRAM. *CAL*, 2015.

[320] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry. RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization. In *MICRO*, 2013.

[321] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*, 2017.

[322] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Gather-scatter dram: In-dram address translation to improve the spatial locality of non-unit strided accesses. In *MICRO*, 2015.

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

[324] V. Seshadri and O. Mutlu. Simple Operations in Memory to Reduce Data Movement. In *Advances in Computers, Volume 106*. 2017.

[325] V. Seshadri and O. Mutlu. In-dram bulk bitwise execution engine. *ArXiv*, 2019.

[326] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *ISCA*, 2016.

[327] J. M. Shalf and R. Leland. Computing Beyond Moore's Law. *Computer*, 2015.

[328] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *MICRO*, 2019.

[329] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks. Co-Designing Accelerators and SoC Interfaces Using gem5-Aladdin. In *MICRO*, 2016.

[330] Y. S. Shao, S. L. Xi, V. Srinivasan, G. yeon Wei, and D. Brooks. Towards Cache-Friendly Hardware Accelerators. In *SCAW*, 2015.

[331] A. Sharma, F. Schuhknecht, and J. Dittrich. Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting. In *SIGMOD*, 2018.

[332] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing*, 1997.

[333] D. E. Shaw, S. J. Stolfo, H. Ibrahim, B. Hillyer, G. Wiederhold, and J. A. Andrews. The NON-VON Database Machine: A Brief Overview. *IEEE DEB*, 1981.

[334] Y. Shen, M. Ferdman, and P. Milder. Maximizing cnn accelerator efficiency through resource partitioning. In *ISCA*, 2017.

[335] D. Shingari, A. Arunkumar, and C.-J. Wu. Characterization and Throttling-Based Mitigation of Memory Interference for Heterogeneous Smartphones. In *IISWC*, 2015.

[336] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*, 2013.

[337] V. Sikka, F. Färber, A. Goel, and W. Lehner. SAP HANA: The Evolution from a Modern Main-memory Data Platform to an Enterprise Application Platform. *Proc. VLDB Endow.*, 2013.

[338] F. Silfa, G. Dot, J.-M. Arnau, and A. Gonzàlez. E-PUR: An Energy-Efficient Processing Unit for Recurrent Neural Networks. In *PACT*, 2018.

[339] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015.

[340] M. D. Sinclair, J. Alsop, and S. V. Adve. Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models. In *MICRO*, 2015.

[341] G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gomez-Luna, S. Stuijk, O. Mutlu, and H. Corporaal. Nero: A near high-bandwidth memory stencil accelerator for weather prediction modeling. In *FPL*, 2020.

[342] G. Singh, J. Gomez-Luna, G. Mariani, G. Francisco, S. Corda, S. Stuijk, O. Mutlu, and H. Corporaal. NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning. In *DAC*, 2019.

[343] R. Smith. Apple's A9 SoC Is Dual Sourced From Samsung & TSMC. `https://www.anandtech.com/show/9665/apples-a9-soc-is-dual-sourced-from-samsung-tsmc`, 2015.

[344] Stanford Network Analysis Project. `http://snap.stanford.edu/`.

[345] J. Stankowski, D. Karwowski, K. Klimaszewski, K. Wegner, O. Stankiewicz, and T. Grajek. Analysis of the Complexity of the HEVC Motion Estimation. In *IWSSIP*, 2016.

[346] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. *ISCA*, 2000.

[347] H. S. Stone. A Logic-in-Memory Computer. *IEEE TC*, 1970.

[348] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, 2005.

[349] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.*, 2013.

[350] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: A Coherent Accelerator Processor Interface. *IBM JRD*, 2015.

[351] R. Sukale. What Are Reflows and Repaints and How to Avoid Them. `http://javascript.tutorialhorizon.com/2015/06/06/what-are-reflows-and-repaints-and-how-to-avoid-them/`, 2015.

[352] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, and R. Nair. Data access optimization in a processing-in-memory system. In *CF*, 2015.

[353] S. Sutardja. The Future of IC Design Innovation. In *ISSCC*, 2015.

[354] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *AAAI*, 2017.

[355] N. Talati, H. Ha, B. Perach, R. Ronen, and S. Kvatinsky. Concept: A column-oriented memory controller for efficient memory and pim operations in rram. *IEEE Micro*, 2019.

[356] X. Tang, O. Kislal, M. Kandemir, and M. Karakoy. Data Movement Aware Computation Partitioning. In *MICRO*, 2017.

[357] TechInsights. Samsung Galaxy S6. `http://www.techinsights.com/about-techinsights/overview/blog/inside-the-samsung-galaxy-s6/`.

[358] J. Teubner, G. Alonso, C. Balkesen, and M. T. Ozsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE*, 2013.

[359] R. Thompson. Improve Rendering Performance with Dev Tools. ttps://inviqa.com/blog/improve-rendering-performance-dev-tools, 2014.

[360] G. Toderici, D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell. Full Resolution Image Compression with Recurrent Neural Networks. In *CVPR*, 2017.

[361] P. Tsai, C. Chen, and D. Sanchez. Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies. In *MICRO*, 2018.

[362] F. Tu, W. Wu, S. Yin, L. Liu, and S. Wei. RANA: Towards Efficient Neural Acceleration with Refresh-Optimized Embedded DRAM. In *ISCA*, 2018.

[363] Twitter, Inc. Twitter. https://www.twitter.com/.

[364] E. Vasilakis. An Instruction Level Energy Characterization of ARM Processors. Technical Report FORTH-ICS/TR-450, Foundation of Research and Technology Hellas, Inst. of Computer Science, 2015.

[365] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan. SCALEDEEP: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *ISCA*, 2017.

[366] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *ISPASS*, 2016.

[367] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *CVPR*, 2015.

[368] D. Šidlauskas, C. S. Jensen, and S. Šaltenis. A Comparison of the Use of Virtual Versus Physical Snapshots for Supporting Update-intensive Workloads. In *DaMoN*, 2012.

[369] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. G. Luna, M. Sadrosadati, N. M. Ghiasi, and O. Mutlu. Figaro: Improving system performance via fine-grained in-dram data relocation and caching. In *MICRO*, 2020.

[370] WebM Project. Hardware: SoCs Supporting VP8/VP9. http://wiki.webmproject.org/hardware/socs.

[371] WebM Project. WebM Repositories — libvpx: VP8/VP9 Codec SDK. https://www.webmproject.org/code/.

[372] WebM Project. WebM Video Hardware RTLs. https://www.webmproject.org/hardware/.

[373] S. Wegner, A. Cowsky, C. Davis, D. James, D. Yang, R. Fontaine, and J. Morrison. Apple iPhone 7 Teardown. http://www.techinsights.com/about-techinsights/overview/blog/apple-iphone-7-teardown/, 2016.

[374] A. Wei. Qualcomm Snapdragon 835 First to 10 nm. http://www.techinsights.com/about-techinsights/overview/blog/qualcomm-snapdragon-835-first-to-10-nm/, 2017.

[375] WordPress Foundation. WordPress. https://www.wordpress.com/.

[376] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine learning at facebook: Understanding inference at the edge. In *HPCA*, 2019.

[377] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *ISCA*, 2013.

[378] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: the architecture and design of a database processing unit. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 255–268. ACM, 2014.

[379] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. S. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *ArXiv*, 2016.

[380] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. Beyond the Wall: Near-Data Processing for Databases. In *DaMoN*, 2015.

[381] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu. Processing-in-Memory Enabled Graphics Processors for 3D Rendering. In *HPCA*, 2017.

[382] Xiph.Org Foundation. Derf Video Test Collection. https://media.xiph.org/video/derf/.

[383] X. Xu, Y. Ding, S. X. Hu, M. T. Niemier, J. Cong, Y. Hu, and Y. Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1:216–222, 2018.

[384] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai. Seraph: An Efficient, Low-Cost System for Concurrent Graph Processing. In *HPDC*, 2014.

[385] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *VLDB Endow.*, 2014.

[386] Yu, Xiangyao. DBx1000. https://github.com/yxymit/DBx1000.

[387] T. Yuanyuan. Building Systems for Big Data Analytics: From SQL to Machine Learning and Graph Analysis. In *BigDas Workshop at SigKDD*, 2017.

[388] S. Zeuch and J.-C. Freytag. Qtm: modelling query execution with tasks. *Proceedings of the VLDB Endowment*, 2014.

[389] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *HPDC*, 2014.

[390] H. Zhang, P. V. Rengasamy, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das. Race-to-Sleep + Content Caching + Display Caching: A Recipe for Energy-Efficient Video Streaming on Handhelds. In *MICRO*, 2017.

[391] H. Zhang, P. V. Rengasamy, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das. Race-To-Sleep + Content Caching + Display Caching: A Recipe for Energy-eficient Video Streaming on Handhelds. In *MICRO*, 2017.

[392] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *HPCA*, 2018.

[393] X. Zhang, J. Li, H. Wang, D. Xiong, J. Qu, H. Shin, J. P. Kim, and T. Zhang. Realizing Transparent OS/Apps Compression in Mobile Devices at Zero Latency Overhead. *IEEE TC*, 2017.

[394] T. Zhao, Y. Zhang, and K. Olukotun. Serving recurrent neural networks efficiently with a spatial accelerator. *ArXiv*, 2019.

[395] 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.

[396] S. Zhu and K.-K. Ma. A New Diamond Search Algorithm for Fast Block Matching Motion Estimation. In *ICICS*, 1997.

[397] Y. Zhu and V. J. Reddi. WebCore: Architectural Support for Mobile Web Browsing. In *ISCA*, 2014.

[398] Y. Zhu and V. J. Reddi. WebCore: Architectural Support for Mobile Web Browsing. In *ISCA*, 2014.

[399] Y. Zhu and V. J. Reddi. GreenWeb: Language Extensions for Energy-Efficient Mobile Web Computing. In *PLDI*, 2016.

[400] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian. Graphq: Scalable pim-based graph processing. *MICRO*, 2019.

[401] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *TIT*, 1977.