Computing Big-Data Applications Near Flash

by

Shuotao Xu

B.S., University of Illinois at Urbana-Campaign (2012)
S.M., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

© Massachusetts Institute of Technology 2021. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by . . . .
Arvind
Johnson Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Computing Big-Data Applications Near Flash

by

Shuotao Xu

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Current systems produce a large and growing amount of data, which is often referred to as Big Data. Providing valuable insights from this data requires new computing systems to store and process it efficiently. For a fast response time, Big Data typically relies on *in-memory* computing, which requires a cluster of machines with enough aggregate DRAM to accommodate the entire datasets for the duration of the computation. Big Data typically exceeds several terabytes, therefore this approach can incur significant overhead in power, space and equipment. If the amount of DRAM is not sufficient to hold the working-set of a query, the performance deteriorates catastrophically. Although NAND flash can provide high-bandwidth data access and has higher capacity density and lower cost per bit than DRAM, flash storage has dramatically different characteristics than DRAM, such as large access granularity and longer access latency. Therefore, there are many challenges for Big-Data applications to enable *flash-centric* computing and achieve performance comparable to that of *in-memory* computing.

This thesis presents *flash-centric hardware architectures* that provide high processing throughput for data-intensive applications while hiding long flash access latency. Specifically we describe two novel flash-centric hardware accelerators, BlueCache and AQUOMAN. These systems lower the cost of two common data-center workloads, key-value cache and SQL analytics. We have built BlueCache and AQUOMAN using FPGAs and flash storage, and show that they can provide competitive performance of computing Big-Data applications with multi-terabyte datasets. BlueCache provides a 10-100× cheaper key-value cache than DRAM-based solution, and can outperform DRAM-based system when the latter has more than 7.4% misses for a read-intensive workloads. A desktop-class machine with single instance of 1TB AQUOMAN disk can achieve performance similar to that of a dual-socket general-purpose server with off-the-shelf SSDs. We believe BlueCache and AQUOMAN can bring down the cost of acquiring and operating high-performance computing systems for data-center-scale Big-Data applications dramatically.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science and Engineering

Acknowledgments

First and foremost, I would like to thank my advisor Arvind for all the research coaching throughput my PhD journey. Arvind is patient and supportive, who is always there for his students whenever they need him. He is willing to spend hours diving deeply with his students to understand their research problems, even if it means that he has to make time from other aspects of his busy work. It was a privilege to learn from Arvind and observe first-hand his constant passion and curiosity to learn new research topics in the field. Arvind's top-down research approach to abstract complex research problems down to simple, systematic and elegant solutions has inspired and influenced me deeply.

I would like to thank my thesis committee members, Prof. Daniel Sanchez and Prof. Adam Belay, for their help and feedback on my research. Although Daniel and Adam are not my academic advisors, they have been providing all kinds of help and brought invaluable new aspects to my thesis. Their help has broadened my horizons on the field of architecture and system research.

I would also thank other CSAIL faculties whom I had opportunities to interact with. In particular, I would like to thank Prof. Joel Emer, and Prof. Mengjia Yan, for their help and advice during my job search.

I am also grateful that I had the special chance to be an TA for 6.004 in Fall 2018. I had the privilege to work with a excellent group of teaching staffs in that semester, including Prof. Arvind, Prof. Daniel Sanchez, Silvina Hanono Wachman and Prof. Song Han. I have learned tremendously how to organize new teaching materials from all of them.

I would like to acknowledge my mentors from industry. I am grateful that Vijay Balakrishnan and Oscar Pinto brought me to San Jose for a summer internship at Samsung, and that Jin Li, Sudipta Sengupta and Jaeyoung Doo offered me the opportunity to intern at Microsoft Research in Redmond.

I am also thankful to have opportunities to work with an excellent group of research collaborators during my graduate study. The BlueDBM team has a special group of highly talented people. I am very grateful that I was able to work closely with Sang-woo Jun, Ming Liu, Sungjin Lee and Chanwoo Chung, and learned aspects of flash storage and hardware

accelerators together with them. I would also like to thank my recent collaborators in our group, Thomas Bourgeat, Tianhao Huang and Xuhao Chen. I would like to thank Thomas for quick and genuine discussions about AQUOMAN, which has influenced AQUOMAN programming model deeply. I really appreciate Tianhao's systematic work on finding all the related work on AQUOMAN, which has helped shape AQUOMAN's overall system angle. And I have learned a lot from Xuhao about graph processing recently.

Moreover, I have been using Connectal for all my projects, and I would like to acknowledge Jamey Hicks and Jon Ankcorn for building and maintaining it. For some reason Jamey is always around Stata center, and I really appreciate his prompt on-prem tech consultation for us.

I am also deeply grateful that I had chances to interact with other CSG members: Sizhuo Zhang, Muralidaran Vijayaraghavan, Andy Wright, Joonwon Choi, Abhinav Agarwal, Nirav Dave, Asif Khan, Richard Uhler, Xiangyao Yu and Guowei Zhang. I really enjoyed all the discussions and knowledge about computer architecture research. In particular, I would always cherish the moments with Sizhuo, Murali, and Ming, when we regularly dined out together on Friday nights and discussed fun stuffs about life and work.

Last but not least, I want to thank my parents and my in-laws for their unconditional support and unwavering love for me. My last year as a PhD student was unconventional because of COVID-19, and I would like to especially thank my wife and our almost one-year-old son for keeping me in good company in Cambridge during these special COVID times.

Contents

1 Introduction **19**

1.1 "Elastic" Clouds for Big-Data Applications 22

1.1.1 Example Applications in "Elastic" Cloud 23

1.1.2 Network Bottlenecks in Disaggregated Clouds 26

1.1.3 Mitigation of the Network Bottleneck 27

1.2 Flash Storage: Opportunities and Challenges 30

1.2.1 Opportunities for High-performance Data Processing with Flash . . 31

1.2.2 NAND Flash Characteristics 33

1.2.3 Legacy Storage Architecture 34

1.2.4 In-Storage Computing 37

1.3 Reconfigurable Hardware Accelerators 38

1.3.1 Addressing General Questions and Concerns about FPGAs 39

1.3.2 High availability of FPGAs in the Cloud 42

1.4 BlueDBM: Near-flash Hardware Accelerator Platform 43

1.4.1 BlueDBM Storage Software Interface 44

1.5 Thesis Contributions 45

1.6 Thesis Organization 46

I Flash-based Key-value Cache for Latency-critical Applications 48

2 Part I Introduction and Background **49**

2.1 Overview and Motivation 49

2.2 A Use Case of Key-Value Store 52
2.3 DRAM-based Key-Value Store 53
2.3.1 Discussion of KVS performance comparison 55
2.4 Flash-based Key-Value Store 56

3 BlueCache Architecture 59
3.1 KVS Protocol Engine 61
3.2 Network Engine 62
3.3 KV-Index Cache Manager 64
3.4 KV-Data Store Manager 66
3.4.1 DRAM KV-Data Store 66
3.4.2 Flash KV-Data Store 67
3.5 From KVS Cache to Persistent KVS 68
3.6 Architectural Innovations 69
3.7 BlueCache Software Interface 70

4 Implementation and Evaluation 73
4.1 BlueDBM Platform 73
4.1.1 FPGA Resource Utilization 74
4.2 Power Consumption 74
4.3 Single-Node Performance 75
4.3.1 DRAM-only Performance 75
4.3.2 Performance with Flash 77
4.4 Multi-Node Performance 79
4.4.1 Operation Throughput 79
4.4.2 Operation Latency 80
4.5 Application Multi-Access Performance 80
4.6 Social Networking Benchmark 82
4.6.1 Experiment setup 84
4.6.2 Experiments 86
4.7 Part I Conclusion 90

4.8 A Retrospective on Hardware-acceleration near Flash for Latency-sensitive Workloads . . . 91

II In-storage Analytic-Query Accelerator for SQL Analytics 95

5 Part II Introduction and Background 97

5.1 Overview and Motivation . . . 97

5.2 Background: Database Accelertors . . . 99

5.2.1 In-storage big data analytics framework . . . 100

5.2.2 General database accelerators . . . 101

5.2.3 Accelerators for certain database operators . . . 103

5.2.4 Query-specific reconfigurable accelerators . . . 103

6 Dataflow map of a query 105

6.1 Single table query . . . 105

6.2 Join – a multiple table query . . . 107

7 Overview of AQUOMAN Architecture 109

7.1 Multi-SSD AQUOMAN . . . 111

7.2 Programming AQUOMAN . . . 111

7.2.1 Software Interface . . . 111

7.2.2 Query-planning for AQUOMAN . . . 113

7.3 Query-planning for AQUOMAN . . . 115

8 AQUOMAN Microarchitecture 117

8.1 Row Selector . . . 118

8.2 Row Transformer . . . 119

8.3 SQL Swissknife . . . 122

8.3.1 Aggregate GroupBy . . . 123

8.3.2 TopK . . . 124

8.3.3 Merger . . . 126

8.3.4 1GB-Block Streaming Sorter 127
8.4 AQUOMAN Memory Management 128
8.5 Suspending Query Processing on AQUOMAN 129
8.6 AQUOMAN as a Near-memory Accelerator 131

9 FPGA Implementation and Evaluation 133
9.1 FPGA Prototype 133
9.2 Query Evaluation Setup 135
9.2.1 Evaluation Data-Set 135
9.2.2 Baseline Setup 135
9.2.3 AQUOMAN Setup 135
9.3 Single-Table Query Evaluations 136
9.3.1 Query A: Filter Operation with High Selectivity 136
9.3.2 Query B: Query with column reuse 137
9.3.3 Query C: Aggregation GroupBy 139
9.3.4 Query D: TopK 140
9.3.5 Discussion on Row Selectivity 141
9.4 Multi-Table Queries 142
9.4.1 AQUOMAN Multi-table Evaluation Methodology 142
9.4.2 Query E: Join 142

10 TPC-H Benchmark with AQUOMAN 147
10.1 AQUOMAN Simulator 148
10.2 Experiment Setup 148
10.2.1 Evaluation Data-set 148
10.2.2 Baseline Setup 149
10.2.3 AQUOMAN Setup 149
10.3 AQUOMAN TPC-H Evaluation 151
10.3.1 Run Time 151
10.3.2 Memory Footprint 152
10.3.3 Advantages of AQUOMAN 153

10.4 Validating simulation results on FPGA 153
10.5 Part II Conclusion 155

11 Conclusion and Future Work 157
11.1 Short-term Future Work 158
11.1.1 Distributed In-storage SQL analytic Offloading 158
11.1.2 Graph-Pattern Mining using Accelerated Flash Storage 159
11.1.3 Machine Learning Accelerators 159
11.2 Long-Term Future Work 160
11.2.1 Make programming flash-centric computing easier 160
11.2.2 Memory-centric computing with emerging memory technology . . 160

List of Figures

1-1 "Disaggregated" Data-center Architecture 22

1-2 BlueDBM System Architecture 43

1-3 BlueDBM Software Interface 44

2-1 Using key-value stores as caching layer 50

2-2 Components of in-memory key-value store 53

2-3 Internal data structures of DRAM-based KVS and flash-based KVS 57

3-1 BlueCache high-level system architecture 59

3-2 A BlueCache architecture implementation 60

3-3 Application to KVS Protocol Engine communication 62

3-4 Network Engine Architecture 63

3-5 4-way set-associative KV-index cache 64

3-6 KV-data store architecture 66

3-7 Log-structured flash store architecture 68

3-8 BlueCache software interface 71

4-1 Single-node operation throughput on DRAM only 75

4-2 Single-node operation latency on DRAM only 76

4-3 Single-node operation throughput on flash 77

4-4 Single-node operation latency on flash 78

4-5 Multi-node GET operation bandwidth. (a) single app server, multiple Blue-Cache nodes, (b) multiple app servers, single Bluecache node, (c) multiple app servers, multiple BlueCache nodes 79

4-6 Multi-node GET operation latency on DRAM/Flash 81
4-7 Accessing BlueCache via software interface, key-value pair sizes are 1KB . 82
4-8 End-to-end processing latency comparison of SSD KVS software and BlueCache 82
4-9 Throughput of memcached for relative network latencies 85
4-10 Performance of BlueCache and other KVS systems as augmentations to a MySQL database, which stores simulated data of a social network generated by BG benchmark. Evaluations are made with various numbers of active social network users. 86
4-11 BG performance for different capacity miss rates, *both memcached and BlueCache have the same capacity of 15GB* 88
4-12 BG performance for different coherence miss rates 89

5-1 Using AQUOMAN for SQL analytics 98

6-1 Aggregate Query 105
6-2 Dataflow of an Aggregate Query 106
6-3 A Join Query 107
6-4 Dataflow of a join query 108

7-1 Overall Architecture of AQUOMAN 109
7-2 System Stack with AQUOMAN 111
7-3 Table Task Structure 112
7-4 JOIN query *Table Task*s 114
7-5 JOIN query data-flow graph 114

8-1 Architecture of Row Vector Selection 118
8-2 Row Transformer Architecture 119
8-3 SQL Query Example for Table Transformation 120
8-4 Data-Flow Execution Diagram of Table Transform 120
8-5 Micro-architecture of a Row Transformer PE 121
8-6 SQL Swissknife Architecture 123

8-7 Aggregate-GroupBy Accel. 123
8-8 TopK Accelerator 125
8-9 Merger Architecture 126
8-10 Streaming Sorter Architecture 127
8-11 AQUOMAN memory operations of a three-way join (*T0.JK0=T1.JK1 and T1.JK2=T2.JK3*) 128

9-1 Query A: High-selectivity Filter: No Column Reuse 137
9-2 Query B: High-selectivity Filter: Column Reuse 138
9-3 Query C: Aggregate-GroupBy Query 140
9-4 Query D: TopK 141
9-5 Sort-Merge Join(AQUOMAN) vs. Merge Join(MonetDB) 144
9-6 Sort-Merge Join(AQUOMAN) vs. Hash Join(MonetDB) 144

10-1 TPC-H SF-1000 AQUOMAN Performance Profiling 150
10-2 TPC-H queries on FPGA prototype 154

List of Tables

1.1 Storage device comparison (prices are representative numbers for 2021 on the Internet) 31

4.1 Host Virtex 7 resource usage 74
4.2 BlueCache power consumption vs. other KVS platforms 74
4.3 KVS storage technology comparison 84

5.1 Representative near-data SQL accelerators 99

7.1 SQL (Sub)Operators 113

8.1 PE Instruction Set 122

9.1 AQUOMAN resource usage on VCU108 134
9.2 Streaming Sorter resource usage on VCU118 134
9.3 1GB-Block Streaming Sorter Throughput 134
9.4 *Table Task*s of Query E 143

10.1 x86 Host and AQUOMAN Disk Setup 149

Chapter 1

Introduction

Current systems generate a vast amount of data continuously in various settings, such as eCommerce [46, 103], social networking [49, 166], web search [107], and sharing economy [39]. For example, in 2018 Uber generated 100 terabytes of trip data daily, and performed analytics on 100 petabytes of collected data [39]. Such a vast and fast growing data is often referred to as Big Data. This collection of Big Data and, the application of data science and machine learning methods to it, has become one of the largest drivers of today's IT industry. However, as the amount of data continues to grow, further progress will depend on our ability to store and analyze it cost-effectively.

Big Data, by definition, does not fit in the main memory of a single server (∼1TB maximum), and typically requires a server cluster to manage and run complex queries on it. Typically such Big-Data applications are run in the "cloud", where *application servers* and *storage servers* are physically separate and connected via a network. In such a "disaggregated" data-center architecture, applications typically need to fetch data from storage servers to the application servers, and then perform computations on the data.

One way to provide fast query responses is by deploying *in-memory* solution, where datasets are kept in DRAM for the duration of computation and large-scale systems are created by aggregating main memories of server machines. Such in-memory solutions are highly popular in the cloud, and are used by many Big-Data frameworks, such as Spark [211, 213]. This RAMCloud [170] style of system organization can provide fast random word-level accesses to datasets and overcome the long latency, large access granularity and

limited bandwidth of central storage, but is only effective when application's working set is captured by the aggregated DRAM capacity. At the same time, this requires a significant amount of hardware resources, CPUs and DRAM, to deliver high-performance Big-Data applications, and results in high cost of equipment, area and power. Meanwhile, if the amount of DRAM is not sufficient to hold the working-set of an application, performance deteriorates catastrophically [208].

Another popular solution to mitigate the slow storage access over network is by offloading computation to storage. Recent research [141] has explored extending high-performance key-value stores with storage function offloading support, which can push and execute code near data with microsecond latency. Analytic workloads using databases has also explored such a solution; however storage servers does not have sufficient CPU and DRAM resources for complex operators, and only simple operators such as *Filter* and *Aggregate without GroupBy* are offloaded in production systems [28, 43, 191]. Near-storage offloading method can offer an order-of-magnitude performance improvements [210], but is still 100× inferior to a share-nothing distributed database where each server has local data access [195]. More aggressive computation push-down would essentially bypass the slow network entirely, but inevitably increases the CPU and DRAM capabilities of storage servers significantly.

In this thesis we explore whether flash storage and hardware accelerators can provide a cheaper and more power-efficient solution to big-data applications in the cloud. Using flash storage instead of DRAM can greatly reduce the high cost of acquiring and operating a large cluster of in-memory computing systems, because flash storage has 100× higher density than DRAM, is 100× cheaper per bit and operates at one fiftieth of DRAM's power. Computing Big-Data applications near flash can potentially bring down the high cost of acquiring and operating a server cluster needed for in-memory computing. Using conventional computing processors for flash-centric computing, such as CPUs and GPUs, can yield satisfactory performance with a much smaller DRAM requirement, but they are still power hungry and consume several hundreds of watts per machine [85, 216] . *Hardware accelerators* can greatly improve performance and efficiency of query processing by specialization. A single hardware accelerator often consumes around tens of watts, and can typically fit within the power limit of a storage peripheral device [126, 127, 129, 208]. If Big-Data applications store

datasets entirely in flash and use hardware accelerators to run queries without degrading performance then the cost of Big-Data applications can be lowered dramatically than the current solutions.

This thesis addresses the challenges of building new *flash-centric hardware architectures* to lower the cost of Big-Data applications with multi-terabytes datasets using FPGAs and flash storage in data-centers. In particular, we focus on two important types of workloads that run in the "elastic" cloud: *latency-sensitive* interactive online services, such as internet searches and social networking, and *analytic workloads* on terabytes/petabytes datasets. For latency-sensitive workloads one typically employs a middle-layer of DRAM-based "caching services" to overcome the long-latency and coarse-grain storage accesses, while for analytic workloads, application servers typically push simple computations, such as filtering, into storage servers to reduce data movement over the network.

We provide alternative solutions for both types of workloads: BlueCache [208], a scalable flash-based key-value store, for latency-sensitive workloads, and AQUOMAN [207], an end-to-end *in-storage* analytic-query offloading solution, which offloads most SQL operators, including multi-way joins, to SSDs.

BlueCache is a 10∼100× cheaper rack-level appliance for data-center-scale key-value cache than in-memory solutions. In BlueCache, all software components including KVS operations, the flash controller and the network are directly implemented in hardware, to guarantee sub-millisecond processing latency. BlueCache can outperform DRAM-based KVS when the latter has more than 7.4% misses for a read-intensive application.

AQUOMAN uses a streaming computation model, which allows AQUOMAN to process queries with a reasonable amount of DRAM for intermediate results. AQUOMAN is a general analytic query processor, which can be integrated in the database software stack transparently. AQUOMAN-augmented SSDs can greatly increase the query offloading capability of storage servers without increasing their CPU and DRAM resources.

The rest of chapter is a deep dive into the background and motivation of using flash storage and hardware accelerator for more efficient Big-Data processing, and is organized as follows:

- Section 1.1 describes the elastic cloud, example workloads which operate in the cloud,

the network bottleneck in the cloud architecture and existing solutions to mitigate the bottleneck.

- Section 1.2 describes the opportunities and challenges of using flash storage as the main memory for Big-Data applications.
- Section 1.3 discusses why hardware accelerators are needed for *near-flash* Big-Data computation, and why we used FPGAs to implement our accelerators.
- Section 1.5 and 1.6 describes the contributions and the organization of this thesis.

1.1 “Elastic” Clouds for Big-Data Applications

Public clouds, such as Amazon AWS [8], Microsoft Azure [22], offers cheaper and more flexible server systems needed for computation and storage of Big-data than “on-perm” or private clouds. Public cloud are “elastic”, where users can customize their general-purpose server resources in terms of CPU/GPU, DRAM, network and storage, and can dynamically add or reduce resources on the fly based on their applications’ requirements. Moreover, major public cloud vendors can support server deployment of ultra-large scale across multiple geo-locations, which is highly infeasible and expensive for a “on-perm” deployment. Therefore many Big-data applications are running on public cloud to maintain a low cost of managing highly elastic server clusters of high quality.

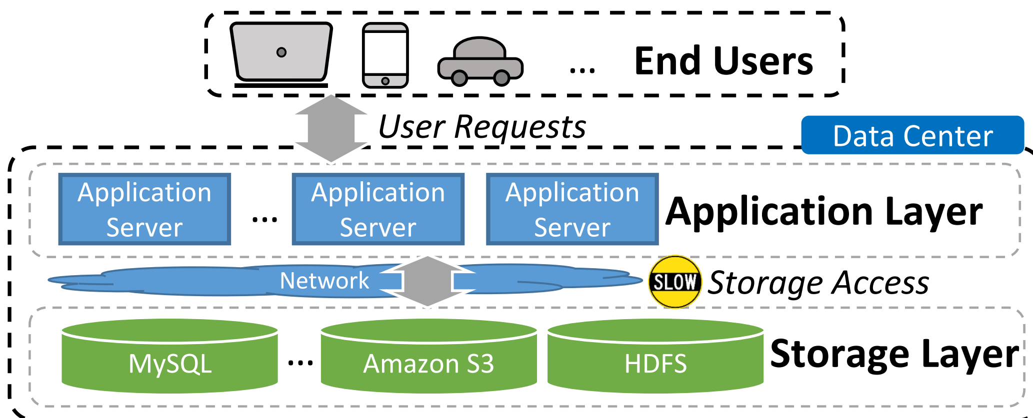


Figure 1-1: “Disaggregated” Data-center Architecture

Modern public cloud are typically employs a "disaggrgated" architecture, where application and storage servers are physically separated and independently managed. As illustrated in Figure 1-1, in such a cloud, the application servers and storage servers are connected via a relatively slow network. This "disaggregated" organization is highly popular because it allows independent compute and storage resources allocations, which increases resources utilization, simplifies server resource management and lower the overall system cost for many big-data applications. For example, many popular data warehousing and analytics systems, such as Presto [26], Snowflake [81], Amazon Redshift [7], are designed purposefully to support running queries on such a "disaggregated" cloud architecture.

1.1.1 Example Applications in "Elastic" Cloud

The emergence of warehouse-scale compute systems, such as public clouds, has brought a large number of big-data applications to the forefront in the form of social networking, web searches, business intelligence, software-as-a-service and more. In general cloud computing platforms aim to provide high-throughput and low-latency computational capability to support high-quality user experience for big-data applications. Yet various big-data applications have their own characteristics, and are concerned with optimizing performance metrics, such as latency and query processing throughput, at vastly different scales. To understand how "disaggregated" cloud architecture can affect the performance of big-data applications, it is important to understand each application's characteristics and its own system design requirements. In this thesis we focus on improving two popular types of big-data applications, *interactive online services* and *analytic workloads*. The rest of this section discusses their characteristics, their typical latency and throughput requirements in system designs, analyses their bottlenecks in the cloud and describes existing solutions to mitigate such bottlenecks.

Interactive Online Services

Large user-facing web services, such as online searches, social networks and e-commerce, are ubiquitous and have become a part of people's normal daily lives. Such applications are

highly interactive, where millions of users actively engage in online activities and expect fluid interactions with the cloud where their queries are serviced. A popular large online service can have hundreds of millions of daily users [167], which puts a significant amount of computational, storage and network-I/O stress to the cloud architecture of today. An interactive online application is typically serviced by a *diverse* mixture of backend services, such as database, recommender systems, advertise systems and more. Typically the backend services organize some indexed data-structures on large data sets to quickly identify the objects of interests for each query, to avoid scanning the entire raw data. Interactive online services need its cloud computing infrastructure to provide high-performance services of extremely high *throughput* and low *latency* to ensure fast and engaging user experiences.

1) Millions-Per-Second Request Rate: Because of such a large number of interactive online services, an interactive online application can generate a large number of end-users requests, which ranges from hundreds of millions to billions of requests per second [152,153]. Such a high request rate typically puts an extraordinary amount of query processing pressure to back-end storage layer, where complex software such as relational databases manages user datasets persistently on disks.

2) Millisecond-scale Tail-Latency: Low latency is critical for the interactive online services. Based on a recent report by Akamai, a 100-millisecond delay is significant, which can reduce customer's conversion rate by 7% and impacts web services' business revenue negatively [5]. Many applications can tolerate milliseconds of latency to provide high-quality interactive online activities for end users. Such user-facing online applications are especially concerned with *tail latency*, such as $99^{th} percentile$ (P99) of the end-user requests. For example, Microsoft online news and retail software stores need deliver fast user experience with 150-millisecond P99 latency [58]. Such a consideration in performance metrics, in addition to throughput and average latency, is as a result of how each end-user request is processed. A single end-user request results in *multiple* queries to a mixture of backend services, and a request is incomplete until *all of its backend requests* have finished. Achieving low tail-latency for data-center-scale computing is challenging, because components in a large-scale system have high degree of variance, and user request traffic changes rapidly, which requires systems to quickly adapt to new situations by redistributing

various computing resources on short notice [84]. To provide low end-user latency for fluid end-user experiences essentially means to mitigate the effects of backend query latency variability and to make sure that the majority of back-end queries are completed within a certain latency bound.

Analytic Workload

Complex analysis of "Big-data" can provide tremendously valuable insights for many domains, such as business intelligence [192], social studies [49], biology [23, 157], personal/public health [15] and more. Typically a large user-facing online service can generate several terabytes of user activity data on a daily basis [39], which is first stored in various online transaction processing (OLTP) systems such as MySQL and Postgres databases. Every once in a while (*e.g.* every 24 hours), data stored in a diverse mixture of OLTP services is ingested into a "data lake" or "data warehouse", which unifies the content, format and metadata of all the datasets that need to be analyzed in one place. Typically a "data warehouse" is massively multi-tenant service hosted by a distributed file system, such as HDFS [16], where datasets are stored in compressed columnar format (*e.g.* Apache Parquet [10]) because of better space and processing efficiency. Data analysts run analytical programs via distributed analytical engines (*e.g.* Spark [212] and Presto [26]), which fetches relevant data from the data warehouse from the storage layer, and performs computations using a server cluster in the applications layer. A significant number of analytic queries are expressed as SQL statements, which perform database operations such as filter, join and aggregate. *Analytical workloads* have system design goals with drastically different performance measurements than those of *interactive online services*.

1) 10-100 Concurrent Requests: Unlike interactive online services with millions to billions of active users, analytic workloads are typically internal service within a company, where several business functions and organizational units share the access to the data-sets in the data warehouse. Typically hundreds to thousands of professional data analysts can access the data warehouse [39], which can result in tens to hundreds of concurrent analytic queries in-flight [26].

2) Seconds-Hours Query Latency: Many analytic queries are exploratory and dynamic,

which can have no index pre-built for fast data retrieval, therefore need to scan raw data files to generate a result. Because of this, analytic workload generally take longer to process a query than interactive online services do. Analytic queries have different ranges of latency depending on the amount of data which are being examined. Interactive queries typically examines a relatively small amount of data ($\sim$50GB-3TB compressed), which are sometimes canceled after initial results are returned [26]. Such queries are aimed at gaining quick intuition, which typically take several seconds to 30 minutes to run [26]. Other queries need to provide deep insights on complete and accurate datasets, which can read tens of terabytes of input data and take several *hours* to complete [26,129].

1.1.2 Network Bottlenecks in Disaggregated Clouds

"Disaggregated clouds" greatly reduce the cost of data-center resource management, but the network that connects application servers and storage servers can present as a major system bottleneck. To perform queries on data-sets, application needs to bring data from the back-end storage layer over the network, and then apply computation and service the end-user request. The network access can be slow both in terms of latency and bandwidth.

Latency Overhead

Although data-center networks have improved greatly, applications still suffer from long remote storage access latency. For example, remote storage accesses to Amazon Simple Storage Service (S3) or Google's cloud storage have latencies from tens to hundreds of milliseconds on average [1,83]. In comparison, NAND flash disk access over local PCIe link can be as low as 23 microseconds [20], and high-speed key-value store device can support average access latencies from 150 to 300 microseconds [115]. Therefore a disaggregated cloud architecture can have more than 100$\times$ access storage latency overhead, measured in hundreds of milliseconds.

More importantly, storage *tail* latency over the network, e.g. 99th *percentile* latency, can be *an order-of-magnitude* slower than the average access latency [84]. Therefore it quickly becomes highly challenging for *interactive online services* to keep the tail latency

distribution short, as the complexity and the size of the system scales up and the number of user requests increases [17].

Analytic workloads are generally less sensitive to the latency overhead since query completion times are measured in minutes to hours instead of milliseconds.

Bandwidth Bottleneck

In a disaggregated architecture, *analytic workloads* need to scan terabytes of datasets, which is bounded by the network throughput between the application and storage layer. Typically the internal bandwidth of a storage server is much higher than the external network bandwidth over which data are accessed by application servers. For example, traffic to and from Amazon AWS S3 storage server can now take advantage of up to 25 Gbps (3.125GB/s) of bandwidth [55]. Such a external network bandwidth is roughly less than half of a single high-speed NVMe SSD's bandwidth (*e.g.* 7GB/s) [30]. In addition a 2U storage server blade can typically support 24 such high-speed storage devices [14]. Although high-speed network such as 400GbE may hit the mainstream soon, which can mitigate the interconnect bandwidth bottleneck in the data center [3], moving a large amount of data around via data-center network can still consume a significant amount of power [61].

Because of this enormous gap in local and remote storage access bandwidth in current systems, it has been shown that *analytic workloads* (e.g. SQL analytics), which perform query processing in a disaggregated architecture, can suffer more than *two orders-of-magnitude* performance degradation than in a conventional shared-nothing architecture [195].

Interactive online services can be also suffer from limited bandwidth because of large amount back-end requests associated with each end-user request; however recent work [59, 189] indicates that processing small network packet, which are typical for interactive applications, can quickly become a bottleneck before network bandwidth are depleted.

1.1.3 Mitigation of the Network Bottleneck

Two intuitive solutions exist to mitigate the network bottleneck in a disaggregated cloud: *DRAM-based caching* and *near-data computing*.

DRAM-based Caching for Interactive Online Services

DRAM accesses is much superior than storage server accesses both in terms of access latency and throughput. Local DRAM access can support 100-nanosecond latency and hundreds of gigabytes per second bandwidth. Remote direct DRAM access (RDMA) can support 2 microsecond latency and 100Gbps bandwidth per NIC port [131], which is still several orders-of-magnitude faster than remote storage access. With DRAM-based caching, the application server load the data from the storage server once, cache it in main memory, and then reuses it many times to amortize the high cost of data retrieval from the back-end storage. As a result, an application can improve its performance significantly if its working-set can be cached effectively in DRAM for the duration of computation.

Big-data, by definition, cannot fit within the main memory of a single general purpose machine. Therefore a distributed server cluster needs to be built to aggregate enough DRAM capacity to provide enough cache capacity for applications. Such a RAMCloud system can aggregate 64TB DRAM storage capacity over 1000 servers connected via high-speed network [169, 170]. Compared to a distributed disk-based storage cluster of similar scale, RAMCloud can provide 100× to 1000× better performance.

Interactive online services can have high *temporal* and *spatial locality* in end-user queries, because users in similar region typically have similar interests at a similar time. Because of this, DRAM-based caching is a highly amenable solution to improve the performance of interactive online services, by reducing load to the backend services significantly and decreasing query processing tail latency greatly because of fast data retrieval from in-memory storage. One of DRAM-based caching systems for large-scale datacenter applications are distributed *in-memory* key-value stores (KVS). They are used to augment the slower backend persistent storage, and are highly popular due to its simplicity and effectiveness. To use such a KVS cache, an application server transforms a user read-request into hundreds of KVS requests, where the key in a key-value pair represents the query for the backend and the value represents the corresponding query result. Facebook's memcached cluster [167] and the open-source project Redis [27] are good examples of such a data-center caching systems. In 2008 Facebook maintained over 28 TBs of memcache distributed cache running over

800 servers in its data center to cache billions of objects and support billions of requests per second from over a billion users around the globe [42].

DRAM-based caching solutions can mitigate the slow storage access overhead greatly, but is extremely expensive in terms of energy consumption, price per bit and storage density, which limits its widespread use except for latency-critical interactive workloads with high user-request rates. Moreover it has been observed repeatedly that applications' performance deteriorates significantly if the a significant portion of the working-set cannot fit in DRAM [127, 208].

Near-data Computing for Analytic Workloads

Historically near-data computing has been motivated by two different problems: 1) insufficient DRAM to hold the buffer pool and 2) slow I/O to move the data from storage to DRAM. Two techniques have been used to tackle small DRAM challenges. One is to partition the input data-set and run the query on each partition [88]. Partitioning reduces random disk I/Os at the cost more sequential disk accesses. The other way is to push computation to disks to avoid bringing data into DRAM, which reduces I/O traffic. This idea was explored as early as in the 80's [87]. The primary motivation for early work for near-data computation was to avoid moving data through slow storage network.

In recent years, similar ideas of near-data computation are widely used in production systems in the cloud [28, 105], because of the network bandwidth bottleneck caused by compute and storage separation needed for an elastic cloud. The primary classes of applications in the cloud that offload computation near data are *analytic workloads* because of a large amount of data needed to be processed by each query. Typically simple operators, such as filtering and aggregate, are offloaded in storage because they requires only a reasonably small amount of computation power and DRAM capacity to perform computation of such operators. Examples include IBM/Netezza [191] and Amazon Aqua [43], which pushes *filtering* and *aggregate* to reconfigurable fabrics near storage.

Amazon S3 Select is one of the most popular examples that uses this idea of near-data computing in the cloud. In 2018 AWS released a new service called S3 Select [28] which pushes a limited set of SQL operators to the storage nodes which possesses the data

persistently. Standard S3 API exposes `GET` and `PUT` operators which reads and writes the entire data objects or part of it based of byte offset ranges. S3 select adds three simple SQL operators to the standard S3 API, which offloads *filtering, projection, aggregate without groupby* to storage servers. A recent study shows that computation push-down using S3 Select can improve analytic query's performance by 6.7× compared to a baseline without near-data capabilities [210].

Offloading simple operators such as filtering is effective, however to achieve query performance similar to that of conventional share-nothing databases, more database operators needs to be pushed to the storage servers [195]. In order to accommodate more aggressive offloading, including multi-way join, storage servers need to increase its CPU and DRAM capability significantly to manage large complex data structures stored as intermediate results in memory. Our micro-benchmarks have shown that 11 HW threads and 128GB DRAM are needed to perform hash-join and saturate 2.4 GB/s of disk bandwidth for TPC-H queries. Such an aggressive pushdown approach using general-purpose computing can increase the cost of storage servers dramatically. Meanwhile it also goes against the design principles of such a disaggregated architecture, which separates compute and storage resources.

1.2 Flash Storage: Opportunities and Challenges

One of the most important aspects for high-performance cloud computing systems for "Big-data" is fast access to large datasets. Most platforms today support complex queries on large datasets using sophisticated software packages running on general-purpose machines. For a fast response time, such systems typically rely on *in-memory* computing, which stores datasets in DRAM for the duration of the computation. Since a single high-end server is limited to 0.5TB∼1TB of DRAM, this approach requires a cluster of machines to accommodate multiple terabyte-size data-sets at a high cost of power, space and equipment. Additionally, it has been observed that the performance plummets if working-set size of a query exceeds the total DRAM size.

Recent advances in storage technology have given a new push to process complex queries with high-bandwidth Non-Volatile Memory (NVM). As illustrated in Table 1.1, NAND

flash costs 100× less per bit, consumes 100× less power and offers two orders of higher storage density than DRAM. These characteristics makes flash-based computing systems a cheaper and more power-efficient alternative to *in-memory* computing systems for Big-data applications.

Table 1.1: Storage device comparison (prices are representative numbers for 2021 on the Internet)

Properties	Samsung 980 PRO Series [31, 32]	Samsung 870 EVO Series [29]	Samsung M393AAG40M3B-CYF [2, 33]
Storage Technology	3D V-NAND PCIe NVMe	3D V-NAND SATA SSD	DDR4-2933MHz ECC RDIMM
Capacity *(GB)*	2000	4000	128
Bandwidth *(GB/s)*	7.00	0.56	23.37
Power *(Watt)*	3.51	2.5	15
Price *(Dollar)*	379.99	479.99	1710.00
Price/Capacity *(USD/GB)*	0.19	0.12	13.36
Power/Capacity *(mWatt/GB)*	1.76	0.625	117.19

1.2.1 Opportunities for High-performance Data Processing with Flash

Storage I/O bandwidth plays a critical role in overall data processing performance, which even affects the efficiency of in-memory computing systems [217]. The bandwidth evolution of NAND-flash-based storage over the last decade has greatly outpaced the ability for CPUs to move data from DRAM by 13X [182].

The primary source of the high bandwidth of modern flash storage drives are multi-chip parallelism. In solid-state drives, NAND flash chips are typically organized with a two-level hierarchy, channels and banks. Each channel essentially is a independent bus, and flash-chip commands and responses can be intertwined completely across different flash channels. On each flash channel there are *multiple* flash chips or banks that are connected to the same bus.

Because of multi-bank parallelism, flash commands and responses on each channel can be performed in a non-blocking manner, which overlaps the latency of access between flash command and actual data movement.

This two-level architecture of flash chip organization is able to provide *scalable* flash bandwidth inside the storage device. For example, a sixteen-channel and single-bank flash drive, which is fairly common in production systems nowadays, can provide 6.4 GB/s of throughput [135]. In 2017 the fastest SSD can provide 13GB/s of sequential read bandwidth [34], which is comparable to that of a single DRAM module, such as DDR4-1600 (12.8GB/s). A single dual-socket server could fit 8-16 DDR4 DIMMs while a high-end storage server can fit 20-40 high-speed SSDs, which means that the total storage bandwidth can be similar to or even surpasses the total DRAM bandwidth in a single server.

Another benefit of the two-level architecture is that *random* accesses to flash chips is as fast as *sequential* accesses, while traditionally random disks seeks can be orders-of-magnitude slower than sequential seeks for rotating hard drives.

Such a bandwidth trend could even accelerate with the advent of emerging non-volatile memory, such as Intel's Phase-Change Memory technology [19]. And there are a tremendous opportunities to use flash storage for *analytic workloads*, whose performance is bounded by the dataset access throughput.

Disks used to be regarded as slow secondary storage devices which can have hundreds of milliseconds of access latency. Flash-based storage devices has pushed the access latency envelop to 26 microseconds in 2021 [30]. Although flash access latency is still 1000$\times$ longer than DRAM access latency ($\sim$20-35 ns), microsecond-scale latency has made flash storage a viable choice for latency-critical applications, such as *interactive online services*, which can tolerates 1-100 milliseconds latency (See Section 1.1.1).

However there are many hurdles that flash-based computing systems have to overcome to reach the performance of DRAM-based systems. Flash-based drive are designed as drop-in replacements of conventional hard drives, not DRAM. There are attempts to use as persistent memory plugged in DIMM slots of a general-purpose machine [21], but such approaches can have high inefficiency because a conventional processor pipeline cannot hide the three orders-of-magnitude longer access latency than DRAM. Many challenges

of using flash storage as the primary memory for big-data applications are because of the unique characteristics of NAND flash memory.

1.2.2 NAND Flash Characteristics

Large Access Granularity

NAND-flash are read and written in *page* granularity, which is typically 4KB or 8KB in size [70]. Large access granularity can create serious bandwidth issues for storage devices for fine-grained work-level random accesses, which are often referred as *read* and *write* amplifications.

- *Read Amplification:* Small random reads can *amplify* the effective read I/O to flash. For example, if only an 8-byte word are read from 8KB flash page, the effective flash read bandwidth requirement is amplified by a factor of 1024.

- *Write Amplification:* Random in-place small updates to flash pages can cause even more performance drop than small-sized reads. To update an 8-byte word to a page, the entirety of an 8KB old page has to be read, merged with the new update, which are then written to a new page location. Because of such an extreme level write amplification, some work [70, 144, 148, 181] *appends* updates and new writes to a log on NAND-flash sequentially, which are later *merged* with old flash pages through a background process called *garbage collection*.

Compared with NAND flash, DRAM can be read and written in much smaller units of cachelines, which is typically 64 bytes. Such a 128× differential in access-granularity between flash memory and DRAM can lead sharp performance degradation of applications, if NAND storage are naively used as a extension of main memory.

Complexity and Side-effects of Flash Write

Flash writes can be performed *only* to pages that are previously *erased*, which is a even costlier operation than flash write itself. Pages are erased by a even coarser granularity of

blocks, which typically consists of 128-256 continuous pages, and are performed in several milliseconds [160].

Flash writes and erasures can also introduce lifetime issues of flash drive because they can cause physical wear of the storage cells. As flash cells are erased and over-written repeatedly for many cycles, the bit error rate (BER) of each flash page also increases. Because of increasing BER, each flash page typically have extra bits reserved for error-correction code (ECC) to extend the usable life-cycle of each flash page. Flash pages become unusable if they cannot be error-corrected or erased successfully.

1.2.3 Legacy Storage Architecture

NAND-flash-based SSDs have been developed largely as a higher-performance drop-in replacement of rotating disks, which require special SSD firmware called *flash translation layer* to hide NAND-flash characteristics, and provide a similar storage interface as hard drives. In addition conventional host storage stack were designed for slow I/O devices, which can become a bottleneck for general-purpose machines to fully exploit high-speed storage I/O.

NAND Flash Translation Layer

In order to provide a view that is consistent with a conventional hard drive, commercial NAND-flash-based SSDs uses special software and hardware inside the storage device, which is called a *Flash Translation Layer* (FTL). FTL provides a familiar *Logic Block Address* view for SSD as a drop-in replacement of hard drives, which translates address mapping from logical to physical space and handles many flash issues, such as the complexity and side-effects of writes. Typically, an FTL is a software running on embedded processor inside a SSD, and provides several important functionalities.

1. *Address Mapping:* FTL manages mapping from a logical address space to physical address space in flash chips, because a) flash pages can become unusable after a certain number of write cycles; b) an updated page can be moved and mapped to a new physical location.

2. *Garbage Collection:* Page updates can cause *invalidation* of old pages, which are then marked by the FTL. When the number of invalid pages become large, FTL performs garage collection. The FTL selects a *block* with a only a few valid pages, moves the valid pages in new locations, changes it address mapping, and erase the old block for future use. FTL can run sophisticated algorithm to minimize write amplification, such that the valid pages in a block to during garbage collection is small [148].

3. *Wear-Leveling:* FTL also performs wear-leveling management for flash, to fairly distribute writes and erasures evenly across all physical storage cells. It ensures that NAND flash blocks don't fail prematurely because of a high number of erasure cycles.

4. *Error Correction:* When a page has bit errors, FTL corrects the errors with the error correction code (ECC), such as Reed-Solomon Code [44, 160] and Hamming ECC algorithm [109]. Each NAND flash page has extra space of a few bytes (*e.g.* 32 bytes) by default, where a checksum generated by a ECC algorithm is stored.

5. *Bad-Block Management:* It is normal for NAND flash to contain bad blocks of page. A bad block contains one or more invalid bits, so its data reliability can no longer be guaranteed. Blocks can turn "bad" during erase operation or during manufacturing process. FTL can detect such bad blocks, move the data to good blocks and mark them as bad so that they would be no longer be used.

Because of significant size of data structure and computation involved in FTL, a commercial SSD needs significant hardware resources. Embedded ARM processors with 8 or more cores and multi-GB DRAM are commonly used in a NAND flash drive controller to handle the complexity FTL [148]. Going through this additional layer can add a considerable amount of access latency. Because FTL is typically hidden by manufacturers and managed transparently, it can also cause significant variability in access latency, which interferes with guaranteeing a low *tail latency*.

Furthermore, there are duplicated functionalities in flash-optimized file systems and FTL, because of complete isolation of these two entities. Each entity makes their best effort to maximize the performance and longevity of flash drives, which can sometimes work

against each other and lead to sub-optimal efficiency [147]. To overcome such issues, there are recent research endeavors in file systems to remove the overhead of FTL, which use host storage stack to manage raw flash chips directly [111, 148].

Host Storage Stack Overhead

Traditionally, drives are regarded as slow secondary persistent storage, which only has several hundreds of megabytes of bandwidth and several hundreds of milliseconds of latency. Therefore the host storage stack were designed under the following assumptions:

1. The interconnect between host and storage device has more bandwidth than the storage bandwidth available internally;

2. The execution speed of host storage software stack, including filesystem, block-device driver, and etc, is much faster than accessing secondary storage.

Such assumptions no longer hold because of rapid storage technology advancements. As discussed earlier in Section 1.2.1, flash drives nowadays can provide tens of gigabytes of bandwidth, and support tens of microseconds of latency. Because of this, legacy host storage stack can become obstacles for NAND flash memory to reach its full potential.

The storage interconnect in host storage stack can be slow compared to the aggregate internal bandwidth of NAND flash chips. Although the state-of-art interconnect, PCIe gen4, can provide 2GB/s bidirectional data transmission speed per lane, this still remains insufficient to keep up the bandwidth scaling of flash drives [182]. Because of storage density, pin limitations and cost restrictions, most commercial NVMe drives (*e.g.* products from Samsung [31] and Intel [19]) use four PCIe lanes as a standard, which implies 8GB/s maximum duplex transfer speed. Such an external interconnect speed can be easily transcended by the internal bandwidth provided by massively parallel NAND-flash chips [34]. The evolution of storage interconnect has remained stagnant, which only had two major updates in 2007 and 2017. Although PCIe evolution has picked up its pace recently and PCIe gen5 is just around the corner [9], the storage interconnect may still remain relatively slow compared to storage drives, because emerging non-volatile memory, *e.g.* Resitive RAMs, have even higher bandwidth than flash [66]. Such new storage devices are rapidly

becoming widely accessible [19]. Even if storage interconnect bottleneck disappears in the futures, conventional computing systems still face obstacles in achieving high power efficiency, because a significant amount of energy is needed to move data quickly between storage cells and the processor registers in order to perform computation [61].

On the other hand, because storage access latency has been significantly reduced, execution of the host storage stack is becoming significant in comparison. PCIe bus can have 1-20 microsecond round-trip latency [136, 152], and host-side software can add another 20 microsecond or more latency [66, 68]. In comparison an SSD in 2021 can provide 26 microsecond latency [30]. Recent work in kernel-bypass I/O has improved storage software stack with tens to hundreds of microsecond latency [99, 137, 168]. However, future generations of NVM drives can potentially deliver sub-microsecond latency [66], which could be beyond the limit of the most sophisticated host storage-software-stack.

1.2.4 In-Storage Computing

In-storage computing (ISC) has been explored as earlier as 1980s by researchers in Wisconsin [87]. There main motive was to reduce data movement from storage to compute. Early work has suggested adding processors to disks to do simple filtering [53, 149, 186], however the performance improvement did not justify the cost of adding special-purpose hardware at that time.

As data-sets become larger and hit "memory wall" more frequently, there is a revival of interests in *in-storage computing* in research community [126, 140, 182, 187], to overcome the aforementioned system bottlenecks in legacy host storage stack . Such a resurgence of ISC research are also encouraged by great improvements of VLSI technology, which dramatically brought down the cost of chip designs and integration.

The primary motivation of recent ISC work is to address the storage interconnect bottleneck by offloading computations, such as filtering, which can have major reduction in I/O traffic by producing a much smaller result than raw input data. By computing *near* storage, ISC can also utilize the full internal storage bandwidth, bypassing the slow external interconnect [78, 120, 140, 177, 197]. In addition to higher bandwidth utilization, work [115]

shows that ISC can also deliver lower and less variable access latency, by completely bypassing the host storage OS stack.

A number of ISC research projects [66, 89, 120, 140, 187] use the embedded ARM processors that run FTL software and are already in the SSDs. They show some promising results in offloading simple operators, such as filter. However, such performance gain quickly disappear when more complex computation is offloaded to such power-constrained processors.

Another line of ISC research is to integrate *FPGAs* in storage, where specialized hardware accelerators for more complex computation can be deployed near data [126, 129, 182, 203]. Industry production systems, such as IBM/Nettezza [191] and Samsung's SmartSSD [35], have also explored the use of FPGA-based accelerators near data. Reconfigurable hardware can deliver much higher performance than embedded processor [182], and provides much greater flexibility than ASIC accelerators.

1.3 Reconfigurable Hardware Accelerators

Because of the end of Dennard scaling, performance improvement of multi-core CPU has slowed down [94], which makes it increasingly difficult to keep up with the ever-increasing high speed of secondary drives (See Section 1.2.1). Although ASIC accelerators can address provide much higher performance and power-efficiency than CPUs by specialization [73, 75, 76, 92, 158], they lacks the flexibility that general-purpose processors have. More importantly, the high development cost of ASIC make it difficult to justify their widespread deployments unless when they can accelerate a wide class of applications [123].

Reconfigurable hardware accelerators [142, 159] have become a important technology to continue the performance scaling of general purpose processors, while providing high degree of flexibility and configurability. The most prominent reconfigurable hardware accelerator technology is *Field-Programmable Gate Arrays* (FPGA). FPGAs are composed of many small *programmable* logic blocks and memories, which are all connected via a *reconfigurable* network. Most FPGAs also include a large number of Digital Signal Processing (DSP) blocks, which are hardened ASICs to implement high-performance floating point operations

or matrix-multiplications. State-of-art FPGAs (*e.g.* Xilinx Ultrascale Plus Series) have millions of logic cells, hundreds of gigabits of memory and thousands of DSP blocks [41], which can be used to accelerate a wide range of application spaces from microprocessor simulator [77, 134, 196], machine learning [64, 98, 102, 110, 214, 215], genomics [198], databases [43, 79, 182, 191, 202, 203], networking [152, 179, 189] and more.

The key characteristics of FPGAs that make it more attractive than ASIC are its programmability, and quick adaptability to new applications. An FPGA circuit can be compiled in minutes or hours, and a compiled FPGA image take only several milliseconds to program. This is much more desirable than the long and expensive design and tape-out cycle of ASIC circuit. Meanwhile the new FPGA feature of *partial reconfiguration* allows reprogramming parts of FPGAs to adapt to new application requirements, without deactivating commonly-used hardware IPs, such as PCIe Endpoint, DRAM controller, network and etc. This can increase overall system availability greatly, when FPGAs are integrated in large computing systems (*e.g.* data-center network), such that mission-critical datapath can always be active [179, 189] in FPGA.

1.3.1 Addressing General Questions and Concerns about FPGAs

Although FPGAs have been actively used in the architecture research community, usually software and system people have questions and concerns about FPGAs, especially when to be deployed in production systems. The following paragraphs try to address some of the general questions and concerns about FPGAs raised by software people.

What kind of applications are suitable for FPGA?

There are no simple answers to this question by naming a domain of applications suitable for FPGA. In fact, when an application domain can benefit from specialized circuits significantly, they are typically hardened as ASIC, such as Digital Signal Processing (DSP) units.

To understand which applications are better fitted for FPGAs, we need to understand where performance gains come from for hardware accelerators, which also include ASICs. Generally speaking, hardware accelerators offer more power-efficiency and performance

than CPUs or GPUs because of *specialization*. For example, some machine learning applications may benefit from low bit-precision arithmetic units which are unsupported on CPUs or GPUs. Some applications, such as graph-pattern mining [74], may only issue `load` operations to a shared memory, therefore its hardware accelerator can run without cache-coherency support, and has a much simpler and more scalable memory subsystem than CPU's. Because of specialization, hardware accelerators also have higher power-efficiency than general-purpose circuits, since their circuits only need the logic and memory customized for a specific workload.

Architecturally speaking, hardware accelerators can capture both massive *pipeline-level* and *data-level* parallelism, which are hard to seize simultaneously by CPU or GPU. Through customization, a hardware accelerator has the ability to efficiently capture the inherent parallelism of an application. For example, data-level parallelism can be captured with by replicating homogeneous processing elements as a SIMD unit [176]. Pipeline-level parallelism can be seized by creating deep hardware pipelines that are tailor-made for applications [124, 207, 208]. Applications which expose abundant inherent parallelism of such types can benefit greatly from hardware acceleration.

Compared to ASIC-based, FPGAs offer much higher reconfigurability in addition to all the benefits from hardware accelerations. FPGA-based accelerators become more advantageous than ASICs when

1. An application domain is still evolving rapidly and new functionalities needs to be added frequently, therefore its accelerator architecture is unstable to be a hardened as an ASIC circuit;
2. An application domain can be accelerated significantly by specialized circuits, but are too narrow to justify the high cost of developing ASIC-based accelerators.

FPGAs run at slower clock, therefore it must be slower than ASIC

FPGAs typically runs at hundreds of *megahertz* (MHz), while ASIC circuit runs at one or two *gigahertz* (GHz). Although FPGAs operate at an order-of-magnitude lower frequency than than ASICs, it can still provide competitive processing throughput by more aggressive

pipelining [183]. Although many of the component in FPGAs that are critical for performance are hardened as ASIC (*e.g.* DSPs, high-speeed serial transceiver), and FPGA's "soft" logic only implements control logic which maybe less sensitive to a lower clock frequency.

Meanwhile, newer generation of FPGAs can be clocked at higher frequency because of technology improvement. For example, it is common to clock accelerators around 300MHz frequency on Xilinx's VCU118 FPGA which uses 14nm/16nm technology [207], and some high-performance FPGA implementation can even run at 600Mhz [40].

FPGAs are difficult to program

Programmers write hardware description languages or high-level synthesis code, to assemble generic logic and memory blocks together as a "soft" circuit, which runs custom application-specific acceleration logic. Hardware codes generally can be compiled for different FPGAs vendors, and it is easy to port code from one platform to another when subtle differences are accounted for (*e.g.* different BRAM sizes between Xilinx and Altera). More often than not, high-level design languages(*e.g.* Bluespec System Verilog, HLS) have managed such small subtle differences transparently for programmers.

Although FPGAs can take longer to build and maybe are harder to debug than software, they typically offer much higher performance and efficiency than software running on CPUs. Real-world large-scale FPGA deployment, such as Microsoft Catapult Project [67, 179], shows that FPGAs can be programmed and used in production settings, just as easily and efficiently as software systems.

FPGA accelerators are less area-efficient than ASIC

This is true to an extend because FPGAs uses programmable logic blocks and routing network to map a "soft" logic. The generic logic portion of a circuit, excluding memory, can be 10-20$\times$ bigger on FPGA than ASIC [189]. However, in general, the area of a hardware accelerator is typically dominated by the size of SRAM, which are hard circuits on FPGAs. Other complex circuitry which supports off-chip accesses to peripherals and memory, such as high-speed transceiver for PCIe and NIC, DRAM controller, are also hardened on FPGAs. Modern FPGA silicons are including more and more hardened logic blocks, even embedded

processors, to close the area-efficiency gap from ASICs. In practice, FPGAs take 2-3× larger area than an ASIC with similar functionality [189].

FPGAs are expensive

FPGA market is quite competitive with 2 strong vendors, Xilinx and Altera. Like ASIC and any other silicons, FPGAs' price per unit can be competitive when purchased in large volumes. Two important factors for silicon prices are their yield rate and area. As discussed in previous paragraphs, FPGAs can have 2-3× large area than a special purpose silicon of similar functionality, but are much area-efficient than other dominant silicons in a general-purpose machine, such as CPUs, DRAMs and GPUs. Also FPGAs typically have high yield rate because of their regular structure [189]. Moreover, because of the reconfigurability of FPGAs, the initial equipment cost of FPGAs can be amortized when they accelerate a wide class of applications in a large scale.

1.3.2 High availability of FPGAs in the Cloud

Because of the acceleration and flexibility of FPGAs, there is immense interests in industry in incorporating FPGAs into cloud infrastructures. Two of the largest public cloud providers in North America, Microsoft and Amazon, have deployed FPGAs in a large-scale in their data centers.

Microsoft has deployed all new Azure and Bing servers which have Altera FPGAs built into the network interface card, and are capable of providing sub-microsecond access latency between accelerators in a data center [67, 179]. A large number of workloads are being accelerated by FPGAs in Microsoft Azure, including Host Software-Defined Network [96, 189], machine learning [98], web searches [67, 179] and data compression [97].

Amazon AWS is offering F1 instances with Xilinx FPGAs [6], and provides a hardware shell of base functionalities for accelerator development, as well as complete tool chain for building and programming the FPGAs. Many hardware accelerators have been developed for AWS F1 instances and have show competitiveness in performance and efficiency [183, 198]. Many public clouds in emerging market, such as Alibaba Cloud, are also working with

FPGA vendors to offer FPGA instances to their users [13].

The high availability of FPGA-equipped instances in the public cloud have made accelerating Big-data applications using hardware accelerators practical, with an immerse amount of FPGA resources already in place in data centers, which are easily accessible for developers and researches around the globe.

1.4 BlueDBM: Near-flash Hardware Accelerator Platform

We used MIT's BlueDBM [125, 126, 127] cluster to explore various flash-centric hardware accelerators for Big-Data applications; a detailed discussion of the system can be found in the paper by Sang-Woo Jun et al. [126]

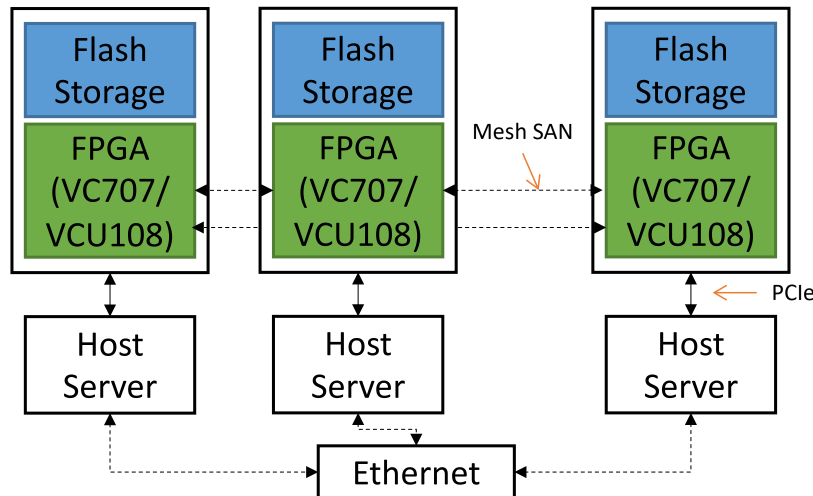


Figure 1-2: BlueDBM System Architecture

BlueDBM is a multi-FPGA platform consisting of identical nodes as shown in Figure 1-2. Each node is a Intel Xeon server with a BlueDBM storage node plugged into a PCIe slot. Each BlueDBM node storage consists of a Xilinx VC707 or VCU108 FPGA development board with two 0.5TB custom flash cards. When BlueDBM was built, the VC707 board is the primary carrier card, which has a Virtex-7 and a 1GB DDR3 SODIMM. Later around 2018, some of the primary carrier cards were updated to VCU108 board, which has a VirtexUltrascale FPGA and 4GB DDR4 DIMM. Each flash card is plugged into a standard FPGA Mezzanine Card (FMC) port on the primary carrier board, and provides an error-free

parallel access into an array NAND flash chips (1.2GB/s or 150K IOPs for random 8KB page read) [160]. Each BlueDBM storage node has four 10Gbps serial transceivers configured as Aurora 64B/66B encoded links with 0.5μs latency per hop. BlueDBM storage nodes are connected via a customized storage area network (SAN) using serial links. BlueDBM also supports a virtual network over the serial links, which provides virtual channels with end-to-end flow control [128].

1.4.1 BlueDBM Storage Software Interface

BlueDBM provides a set of software interfaces to support the execution of existing application as well as modified applications that leverage the in-storage processors in the system. Furthermore, software layers and/or hardware accelerator in BlueDBM must perform flash management functions because BlueDBM uses a raw flash interface in hardware for higher efficiency. The storage software stack is shown in Figure 1-3. User applications can access accelerated flash via four interfaces.

- A raw interface to NAND flash storage via the PCIe driver;
- A block device driver interface;
- A file system interface;
- An accelerator interface over PCIe.

In Figure 1-3, the first three interfaces correspond to the three leftmost arrows. Accelerator interface corresponds to the two rightmost arrows.

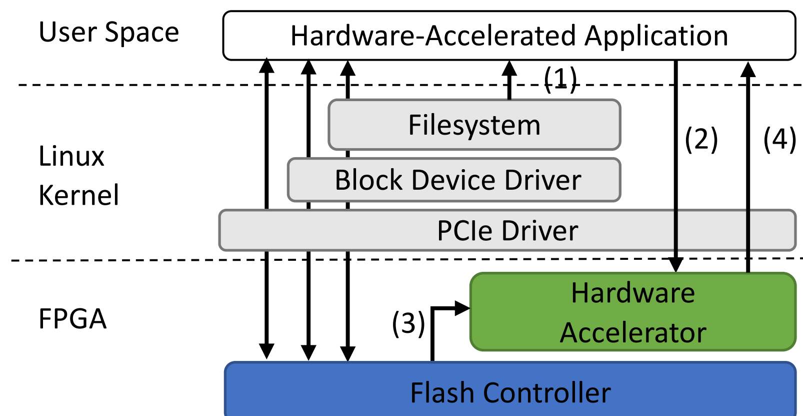


Figure 1-3: BlueDBM Software Interface

Figure 1-3 also shows how user-level applications access hardware accelerators.

1. In the BlueDBM software stack, user-level applications can query the file system for the physical locations of files on the flash. This was made possible because the file system maintains the mapping information to the actual physical locations of the pages.
2. Applications can then provide near-storage accelerator with a stream of physical flash page addresses.
3. Near-storage accelerator can directly access data from flash with low latency.
4. The results are sent to software memory and the user application can be notified via interrupt.

Please note that Step 1 is not mandatory if the flash storage is managed completely by hardware accelerators like in BlueCache. In such a case, user-space application can directly interact with hardware-accelerated storage via an application-specific interface.

1.5 Thesis Contributions

This thesis shows the viability of using cheaper flash-storage instead of costly DRAM for Big-Data applications, by evaluating two important classes of applications, latency-critical applications and analytic workloads, using novel system architecture based on flash storage and in-storage hardware acceleration. The thesis is divided into two parts, where the first part describes BlueCache and the second part describes AQUOMAN.
The following summarizes the contributions of Part I:

1. Design and implementation of BlueCache, a rack-level appliance of a scalable flash-based key-value cache.
2. End-to-end system evaluation that shows BlueCache as an attractive point in the cost-performance trade-off for data-center-scale key-value caches.

Part II makes the following contributions:

1. AQUOMAN, a novel micro-architecture for an in-storage accelerator capable of stream processing a *Table Task* which is a static dataflow graph of operators;

2. Examples of how to derive the dataflow graph of operators to create *Table Task*s for an SQL query, including multi-way joins;

3. A complete in-storage solution which can be fully integrated into a database management software (DBMS);

4. An FPGA-based implementation of AQUOMAN, which can process data stream at the line-rate of our flash controller.

5. An end-to-end evaluation of AQUOMAN using TPC-H benchmarks on 1TB data-set against the baseline of an x86 server with 16 dual-threaded cores and 128GB DRAM.

1.6 Thesis Organization

Part I is organized as follows:

- Chapter 2 movitates the design and implementation of BlueCache, and introduces basic operations of Key-value Caches and discusses and existing research on both DRAM-based and flash-based KVS caches. We also emphasized the importance of a *capacity* cache misses in analyzing the performance of Key-value Cache.

- Chapter 3 describes the architecture of BlueCache, which consists of a homogeneous array of hardware accelerators which directly manage key-value pairs on error-corrected NAND-flash array of chips without any general purpose processor. In this chapter we also describe the software library to access BlueCache hardware, which allows many multi-threaded applications to share BlueCache concurently via simple `GET`, `SET` and `DELETE` APIs.

- Chapter 4 describes a hardware implementation of BlueCache using FPGAs and discusses its implementation platform, hardware resource usage, and power characteristics. In this chapter we also evaluates the raw performance of BlueCache, and show our results of various micro-benchmarks to examine the throughput, latency

and scalablity characteristics of BlueCache. In this chapter we also shows an end-to-end social-networking benchmark that compares BlueCache against other standard software-based KVS solutions as data-center caches.

Part II is organized as follows:

- Chapter 5 motivates the design and implementation of AQUOMAN, and introduces background information and existing research work in accelerating analytical working load using FPGAs and/or ASICs.
- Chapter 6 discusses the anatomy of query processing on database tables, and is followed by examples of translating SQL queries into dataflow maps, or a sequence of *Table Task*s.
- Chapter 7 gives an overview of AQUOMAN micro-architecture, an in-storage generic query offloading machine which maps the execution of a *Table Task*. We also describe the system integration overview of AQUOMAN and its software interface.
- Chapter 8 describes the detailed micro-architecture of AQUOMAN, which consists of three major components, Row Selector, Row Transformer and SQL Swissknife. We also describe several reasons why a query may not be completely processed by AQUOMAN, and discuss that such scenarios of query suspensions on AQUOMAN are uncommon.
- Chapter 9 describes an FPGA implementation of AQUOMAN prototype and shows the results of query evaluations.
- Chapter 10 shows the results of an end-to-end evaluation of AQUOMAN using TPC-H benchmarks on 1TB data-set against the baseline of an x86 server with 16 dual-threaded cores and 128GB DRAM.
- We conclude this thesis and present possible future research directions in Chapter 11.

Part I

Flash-based Key-value Cache for Latency-critical Applications

Chapter 2

Part I Introduction and Background

2.1 Overview and Motivation

Big-data applications such as eCommerce, interactive social networking, and on-line searching, process large amounts of data to provide valuable information for end users in real-time. For example, in 2014, Google received over 4 million search queries per minute, and processed about 20 petabytes of information per day [107]. For many web applications, persistent data is kept in ten thousand to hundred thousand rotating disks or SSDs and is managed using software such as MySQL, HDFS. Such systems have to be augmented with a middle layer of fast cache in the form of distributed in-memory KVS to keep up with the rate of incoming user requests.

An application server transforms a user read-request into hundreds of KVS requests, where the *key* in a key-value pair represents the query for the backend and the *value* represents the corresponding query result. Facebook's memcached [167] and open-source Redis [27], are good examples of such an architecture (Figure 2-1). Facebook's memcached KVS cluster caches trillions of objects and processes billions of requests per second to provide high-quality social networking services for over a billion users around the globe [167]. KVS caches are used extensively in web infrastructures because of their simplicity and effectiveness.

Applications that use KVS caches in data centers are the ones that have guaranteed high hit-rates. Nevertheless, different applications have very different characteristics in

terms of the size of queries, the size of replies and the request rate. KVS servers are further subdivided into application pools, each representing a separate application domain, to deal with these differences. Each application pool has its own prefix of the keys and typically it does not share its key-value pairs with other applications. Application mixes also change constantly, therefore it is important for applications to share KVS for efficient resource usage. The effective size of a KVS cache is determined by the number of application pools that share the KVS cluster and the sizes of each application's working set of queries. In particular, the effective size of a KVS has little to do with of the size of the backend storage holding the data. Given the growth of web services, the scalability of KVS caches, in addition to the throughput and latency, is of great importance in KVS design.

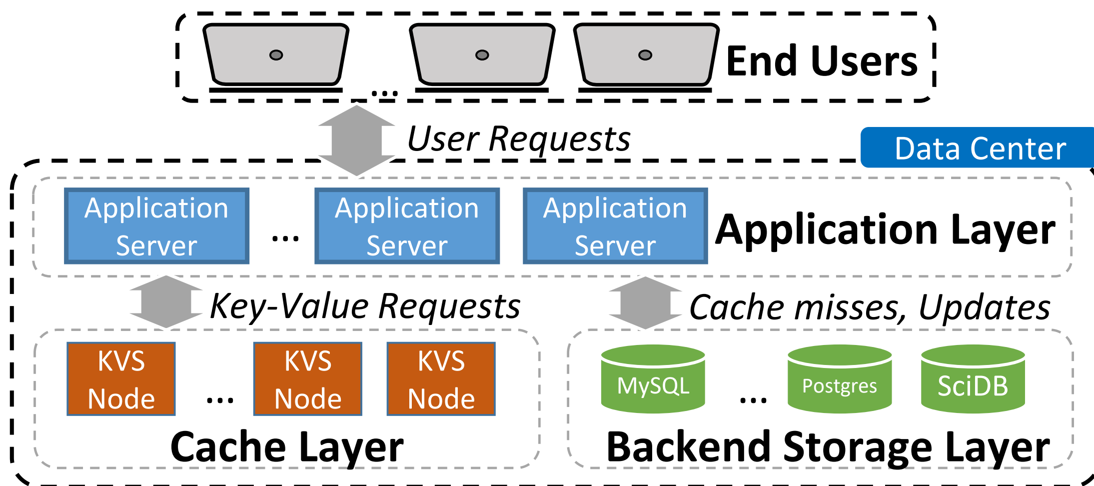


Figure 2-1: Using key-value stores as caching layer

In theory, one can scale up the KVS cluster by increasing the total amount of RAM or the number of servers in the KVS cluster. In practice, however, hardware cost, power/thermal concerns and floor space can become obstacles to scaling the memory pool size. NAND flash is 100× cheaper, consumes 10-100× less power and offers two orders of magnitude greater storage density over DRAM. These characteristics make flash-based KVS a viable alternative for scaling up the KVS cache size. A downside of flash-based KVS is that flash has significantly higher latency than DRAM. Typically flash memory has read latency of 100μs and write latency of milliseconds. DRAM, on the other hand, offers latency of 10-20ns, which is more than four orders of magnitude better than flash. Thus to realize

the advantage of flash, a flash-based architecture must overcome this enormous latency differential. One silver lining is that many applications can tolerate millisecond latency in responses. Facebook's memcached cluster reports 95th percentile latency of 1.135 ms [167]. Netflix's EVCache KVS cluster has 99th percentile latency of 20 ms and is still able to deliver rich experience for its end users [24].

We present BlueCache, a new flash-based architecture for KVS clusters. BlueCache uses hardware accelerators to speed up KVS operations and manages communications between KVS nodes completely in hardware. It employs several technical innovations to fully exploit flash bandwidth and to overcome flash's long latencies:

1. Hardware-assisted auto-batching of KVS requests;
2. In-storage hardware-managed network, with dynamic allocation of dedicated virtual channels for different applications;
3. Hardware-optimized set-associative KV-index cache;
4. Elimination of flash translation layer (FTL) with a log-structured KVS flash manager, which implements simple garbage collection and schedules out-of-order flash requests to maximize parallelism.

Our prototype implementation of BlueCache supports 75X more bytes per watt than the DRAM-based KVS and shows:

- 4.18X higher throughput and 4.68X lower latency than the software implementation of a flash-based KVS, such as Fatcache [199].
- Superior performance than memcached if capacity cache misses are taken into account. For example, for applications with the average query size of 1KB, GET/PUT ratio of 99.9%, BlueCache can outperform memcached when the latter has more than 7.4% misses.

We also offer preliminary evidence that the BlueCache solution is scalable: a four-node prototype uses 97.8% of the flash bandwidth. A production version of BlueCache can easily support 8TB of flash per node with 2.5 million requests per second (MRPS) for 8B

to 8KB values. In comparison to an x86-based KVS with 256GB of DRAM, BlueCache provides 32X larger capacity with 1/8 power consumption. It is difficult to assess the relative performance of the DRAM-based KVS because it crucially depends on the assumptions about cache miss rate, which in turn depends upon the average value size. BlueCache presents an attractive point in the cost-performance trade-off for data-center-scale key-value caches for many applications.

Even though this thesis is about KVS caches and does not exploit the persistence of flash storage, the solution presented can be extended easily to design a persistent KVS.

We will first present the basic operations of KVS caches and discuss the related work for both DRAM-based and flash-based KVS.

2.2 A Use Case of Key-Value Store

A common use case of KVSs is to provide a fast look-aside cache of frequently accessed queries by web applications (See Figure 2-1). KVS provides primitive hash-table-like operations, SET, GET and DELETE on *key-value* pairs, as well as other more complex operations built on top of them. To use KVS as a cache, the application server transforms a user read-request into multiple GET requests, and checks if data exists in KVS. If there is a cache hit, the application server collects the data returned from KVS and formats it as a response to the end user. If there is a cache miss, application server queries the backend storage for data, and then issues a SET request to refill the KVS with the missing data. A user write-request, *e.g.* a Facebook user's "unfriend" request, is transformed by the application server into DELETE requests for the relevant key-value pairs in the KVS, and the new data is sent to the backend storage. The next GET request for the updated data automatically results in a miss which forces a cache refill. In real-world applications, more than 90% KVS queries are GETs [48, 62].

Application servers usually employ hashing to ensure load balancing across KVS nodes. Inside the KVS node, another level of hashing is performed to compute the internal memory address of the key-value pairs. KVS nodes do not communicate with each other, as each is responsible for its own independent range of keys.

2.3 DRAM-based Key-Value Store

DRAM-based key-value stores are ubiquitous as caching solutions in data centers. Like RAMCloud [170], hundreds to thousands of such KVS nodes are clustered togother to provide a distributed in-memory hash table over fast network. Figure 2-2 shows a typical DRAM-based KVS node. Key-value data structures are stored on KVS server's main memory, and external clients communicate with the KVS server over network interface card (NIC). Since both keys and values can be of arbitrary size, an KV-index cache for each key-value pair is kept in a separate data structure. In the KV-index cache the data-part of the key-value pair just contains a pointer to the data which resides in some other part of the cache. This index data structure is accessed by hashing the key generated by the application server. If the key is found in the KV-index cache, the data is accessed in the DRAM by following the pointer in the KV-index cache. Otherwise, the KV-index cache and the DRAM have to be updated by accessing the backend storage.

KVS caches have two important software components: 1) the network stack that processes network packets and injects them into CPU cores; 2) the key-value data access that reads/writes key-value data structures in main-memory. These two components have a strong producer-consumer dataflow relationship, and a lot of work has been done to optimize each component as well as the communication between them.

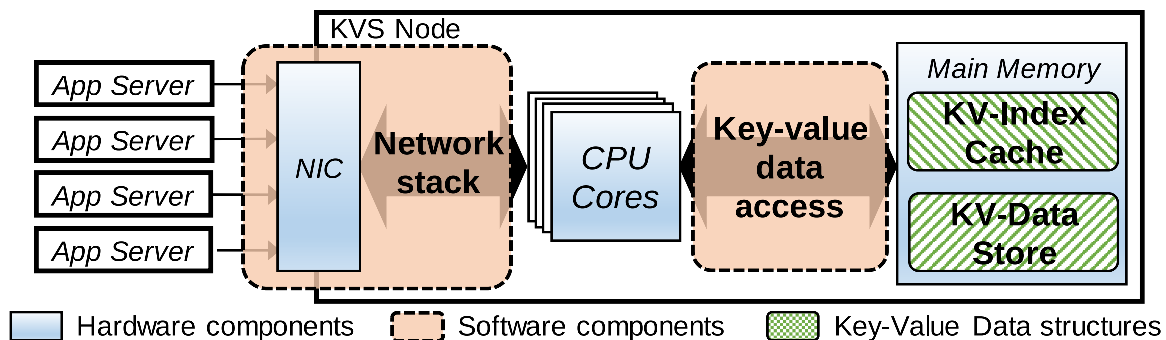


Figure 2-2: Components of in-memory key-value store

A past study [180] has found that more than 90% processing time in memcached is spent in the OS kernel network processing. Researchers have demonstrated as much as 10X performance improvement by having user-space network stack [90, 121, 122, 130, 153,

155, 164, 194]. For example, Jose et al. [121, 122] investigated the use of RDMA-based communication over InfiniBand QDR network, which reduced KVS process latency to below 12μs and enhanced throughput to 1.8MRPS. Another example, MICA [153, 155], exploits modern NIC features such as multiple queues and flow-steering features to distribute packets to different CPU cores. It also exploits Intel Data Direct I/O Technology (DDIO) [116] and open-source driver DPDK [117] to let NICs directly inject packets into processors' LLC, bypassing the main memory. MICA shards/partitions key-value data on DRAM and allocates a single core to each partition. Thus, a core can access its own partition in parallel with other cores, with minimal need for locking. MICA relies on software prefetch for both packets and KVS data structures to reduce latency and keep up with high speed network. With such a holistic approach, a single MICA node has shown 120MRPS throughput with 95$^{\text{th}}$ percentile latency of 96μs [153].

Berezecki et al. [57] use a 64-core Tilera processor (TILEPro64) to run memcached, and show that a tuned version of memached on TILEPro64 can yield at least 67% higher throughput than low-power x86 servers at comparable latency. It showed 4 TILEPro64 processors running at 866Mhz can achieve 1.34MRPS. This approache offers less satisfactory improvements than x86-based optimizations.

Heterogeneous CPU-GPU KVS architectures have also been explored [112, 113, 216]. In particular, Mega-KV [216] stores KV-index cache on GPUs' DRAM, and exploits GPUs' parallel processing cores and massive memory bandwidth to accelerate the KV-index cache accesses. KV-data store is kept separately on the server DRAM, and network packets are processed, like in MICA, using Intel's high-speed I/O [116, 117]. On a commodity PC with two Nvidia GTX 780 GPUs and two CPUs, Mega-KV can process up to 166 MRPS with 95$^{\text{th}}$ percentile latency of 410μs [216].

Researchers have also used FPGAs to offload parts [100, 143, 156] or the entirety [59, 69] of KVS, and demonstrated good performance with great power efficiency. Xilinx's KVS has the highest performance based on this approach, and achieves up to 13.2MRPS by saturating one 10GbE port [59], with the round-trip latency of 3.5μs to 4.5μs. It also shows more than 10x energy efficiency compared with commodity servers running stock memcached.

2.3.1 Discussion of KVS performance comparison

An insightful performance comparison of different approaches is difficult. First of all, all performance results are reported assuming no *capacity* cache misses. Even cache misses due to updates are not properly described. For example, both MICA and Mega-KV experiments that showed the highest performance (>120MRPS) assumed 8B key and 8B value. If the application had 1KB values, then the same cache will hold 128X fewer objects, which should significantly increase the number of capacity misses. Unfortunately, the capacity misses vary from application to application and cannot be estimated based of the number of objects in KVS. Even if capacity misses are rare, the throughput in terms of MRPS will be much lower for larger values. For example, MICA performance drops from 5MRPS to .538MRPS per core if the value size is increased from 8B to 1KB [155]. Moreover, a simple calculation shows that for 1KB values, one 10Gbps port cannot support more than 1.25MRPS.

The second point to note is that higher throughput requires more hardware resources. Mega-KV shows 1.38X more performance than MICA but also increased power consumption by 2X. This is because it uses two Nvidia GTX 780 GPUs which consume approximately 500W [25]. If we look at the performance normalized by the number and speed of network ports then we will reach a different conclusion. For example, MICA shows a performance of 120MRPS using 12 10Gbps Ethernet ports (10MRPS per port), while Xilinx [59] achieves 13.2MRPS using only one 10Gbps port.

One can also ask the question exactly how much resources are needed to keep a 10Gbps port busy. MICA experiments show that 2 Xeon cores are enough to keep up with a 10Gbps Ethernet port [153], and Xilinx experiments show that an FPGA-based implementation can also easily keep up with the 10Gbps port [59]. A past study has pointed out that traditional super-scalar CPU core pipeline of x86 can be grossly underutilized in performing the required KVS computations (networking, hashing and accessing key-value pairs) [156]. The last level data cache, which takes as much as half of the processor area, can be ineffective [156] and can waste a considerable amount of energy. A simple calculation in terms of cache needed to hold all in-flight requests and network packets shows that MICA's

120MRPS throughput can be sustained with only 3.3 MB of LLC, while a dual-socket Intel Xeon processor has a 60MB LLC!

2.4 Flash-based Key-Value Store

There have been several efforts to use NAND flash in KVS designs because it provides much higher storage density, lower power per GB and higher GB per dollar than DRAM [56, 146, 160, 161, 178, 193]. In one organization, NAND flash is used as a simple swap device [138, 185] for DRAM. However, the virtual memory management of the existing OS kernels is not suitable for NAND flash because it leads to excessive read/write traffic. NAND flash is undesirable for small random writes because an entire page has to be erased before new data is appended. High write-traffic not only wastes the I/O bandwidth but shortens the NAND lifetime [50].

A better flash-based KVS architecture uses flash as a cache for objects where the object granularity is one page or larger [101, 154]. Examples of such an architecture include Twitter's Fatcache [199], FAWN-KV [47], Hybrid Memory [174], Xilinx's KVS [60] and FlashStore [85]. KV-index cache stores key-value metadata such as timestamps, which has frequent updates. Like Figure 2-2, they move key-value data to flash while keeping KV-index cache in DRAM, because in-place index updates on flash would make it prohibitively inefficient and complicated. The NAND flash chips are written as a sequence of blocks produced by a log-structured flash management layer to overcome NAND flash's overwriting limitations [147, 148].

A superior way of maintaining an in-memory KV-index cache is to use fixed-size representation for variable size keys (See Figure 2-3). This can be done by applying a hash function like SHA-1 to keys [85, 199] and keeping the fixed-size abbreviated keys on DRAM with pointers to the full keys and values on flash. False positive match of an abbreviated key can be detected by comparing request key with the full key stored on flash. Such data structure organization ensures that 1) on a KV-index cache miss, there is no flash access; and 2) on a KV-index cache hit, a single flash access is needed to access both the key and data.

One can calculate the rarity of a false positive match of an abbreviated key. Assuming

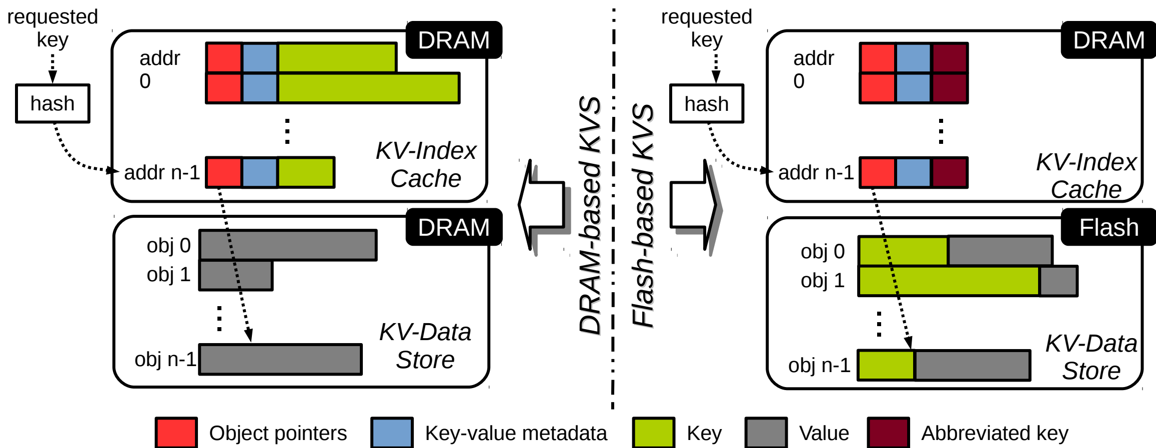


Figure 2-3: Internal data structures of DRAM-based KVS and flash-based KVS

each hash bucket has four 128-bit entries, an 8GB hash table has $2^{33-2-4} = 2^{27}$ hash buckets. Assuming 21-bit abbreviated keys, the false positive for key hit on a index entry is as low as $1/(2^{27+21}) = 1/2^{48}$. This ensures that a negligible portion of KVS misses are penalized by expensive flash reads.

One can estimate the size of KV-index cache based on the average size of objects in the workload. For example, assuming 16-byte index entries, a 16GB KV-index cache can store 2^{30} or ~a billion key-value pairs in flash. Such an index cache can address 1TB KV-data store, assuming average object size of 1KB, or can address 250GB KV-data store, assuming average object size of 256B. Thus, in this system organization, a terabyte of SSD can be paired with 10 to 50 GB of DRAM to increase the size of KVS by 10X to 100X on a single server.

KV-index cache reduces write traffic to NAND flash by 81% [174], which implies 5.3X improvement in storage lifetime. FlashStore [85] is the best performing single-node flash-backed KVS in this category, achieving 57.2 KRPS. Though the capacity is much larger, the total performance, ignoring DRAM cache misses, is more than one to two orders of magnitude less than the DRAM-based solutions. Unlike MICA [153], the throughput of FlashStore is limited by the bandwidth of the flash device and not the NIC. If the flash device organizes NAND chips into more parallel buses, the throughput of flash-based KVS will increase, and it should be possible for the KVS to saturate a 10Gbps Ethernet. The throughput can be increased even further by eliminating flash translation layer (FTL) [80]

in SSDs, as has been shown for flash-based filesystems, such as SDF [171], F2FS [144], REDO [147] and AMF [148].

Chapter 3

BlueCache Architecture

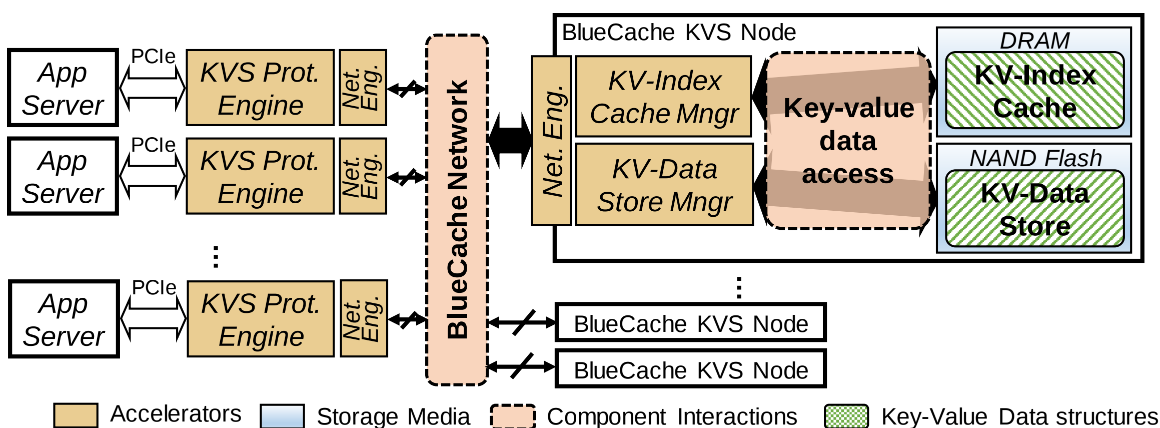


Figure 3-1: BlueCache high-level system architecture

The BlueCache architecture consists of a homogeneous array of hardware accelerators which directly manage key-value pairs on error-corrected NAND-flash array of chips without any general purpose processor (See Figure 3-1). It organizes key-value data structures the same way as shown in Figure 2-3. The KV-index cache stores abbreviated keys and key-value pointers on DRAM, and the KV data, i.e., the full key and value pairs, on flash. BlueCache also uses a small portion of DRAM to cache hot KV-data-store entries. Based on the average size of key-value pairs, BlueCache architecture typically requires the DRAM capacity to be 1/100 to 1/10 of the flash capacity in order to address all the data on flash.

The *KVS Protocol Engine* is plugged into each application server's PCIe slot and forwards each KVS request to the BlueCache KVS node responsible for processing it. In

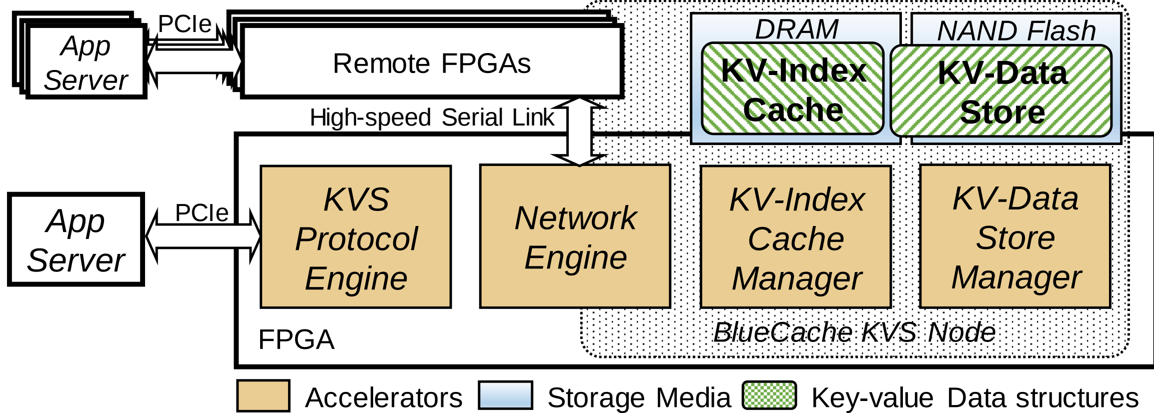


Figure 3-2: A BlueCache architecture implementation

order to maximize parallelism and maintain high performance, KVS requests are batched on application server's DRAM and transferred to the Protocol Engine as DMA bursts. When responses are returned from various KVS nodes, the Protocol Engine batches the responses and sends them back to the application server. The Protocol Engine communicates with KVS nodes via high-speed *BlueCache network* accessed using the *Network Engine*; there is no communication between KVS nodes.

Each KVS node (See Figure 3-1) runs a *KV-Index Cache Manager* and a *KV-Data Store Manager* to manage the KV-index cache and the KV-data store, respectively. Based on the size of KV-index cache, some DRAM is also used to cache KV-data store. Just like the KVS cluster described in Section 2.2, BlueCache architecture deploys two levels of hashing to process a key-value query. The first level of hashing is performed by the KVS Protocol Engine to find the KVS node responsible for the query, while the second level of hashing is performed within the KVS node to find the corresponding index entry in the KV-index cache.

We use Field-programmable Gate Arrays (FPGA) to implement BlueCache (See Chapter 4). In our FPGA-based implementation, each FPGA board has its own DRAM and NAND flash chips and is directly plugged into a PCI Express slot of the application server (see Figure 3-2). Both the KVS Protocol Engine and the KVS node are mapped into a single FPGA board and share the same Network Engine. FPGA boards can communicate with each other using 8 10Gbps bidirectional serial links [128]. In the following sections, we

describe each hardware accelerator in detail.

3.1 KVS Protocol Engine

Unlike memcached [167], which relies on client software (*e.g.* getMulti(String[] keys)) to batch KVS queries, Bluecache uses hardware accelerators to automatically batch KVS requests from multiple applications. The application server collects KVS requests in 8KB segments in the DRAM and, when a segment is filled up, passes the segment ID to the Protocol Engine for a DMA transfer (See Figure 3-3). The application server then picks up a new free segment and repeats the above process. The Protocol Engine receives requests in order and sends an acknowledgement to the application server after reading the segment. A segment can be reused by the application server after the acknowledgement has been received. Since KVS requests are of variable length, a request can be misaligned with the segment boundary. In such cases, the Protocol Engine merges segments to produce a complete KVS request. KVS responses are batched together similarly by the Protocol Engine and sent as DMA bursts to the application server. Our implementation uses 128 segments DMA buffers in each direction.

The *request decoder* (See Figure 3-3) computes the destination BlueCache node ID by using the equation:

$$nodeID = hash(key) \bmod \#nodes \tag{3.1}$$

to distribute the requests across the KVS nodes evenly. The hash function in the equation should be a *consistent hashing function* [133], so that when a node joins or leaves the cluster, minimum number of key-value pairs have to be reshuffled. KVS Protocol Engine also keeps a table of the in-flight request keys because the responses do not necessarily came back in order. In Section 3.3 we will describe how request and response keys are matched to form a response for the application server.

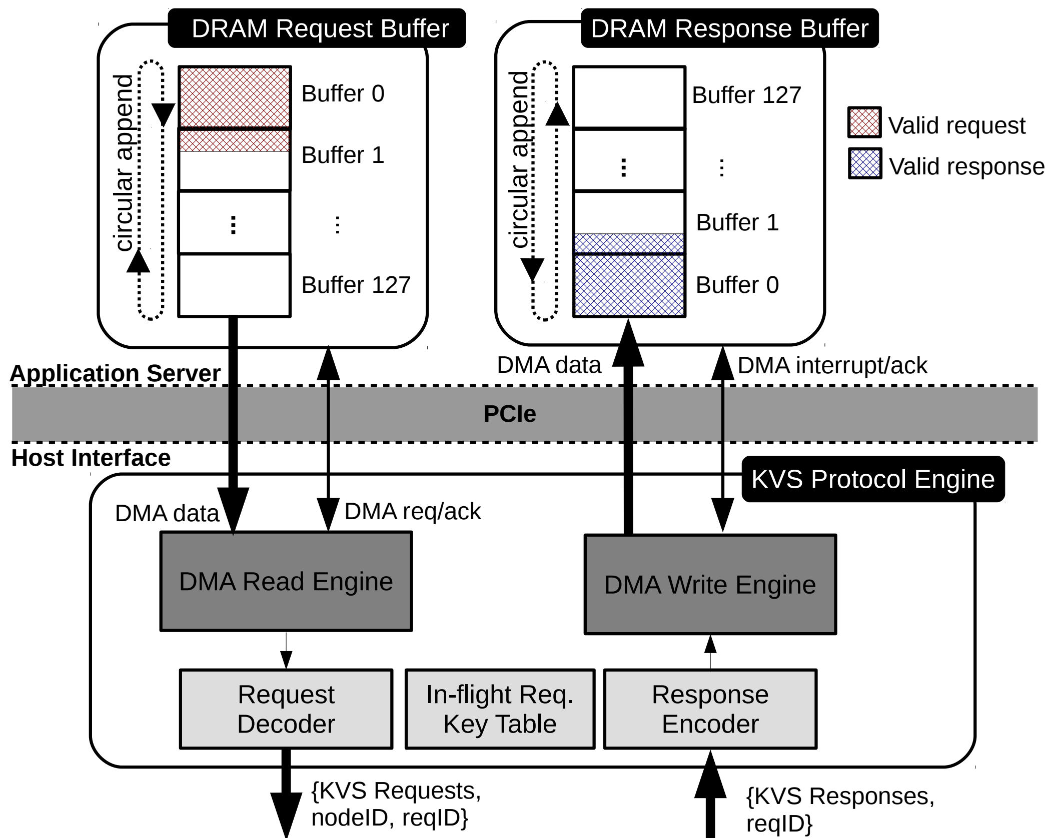


Figure 3-3: Application to KVS Protocol Engine communication

3.2 Network Engine

Each request carries a destination node ID which is used by the Network Engine to route the requests to its destination. Each request also carries the source ID, *i.e.*, the ID of the node that sends the request; these are used by the Network Engine to send back responses. The Network Engine splits the KVS requests into local and remote request queues (Figure 3-4). The requests in the remote-request queue are forwarded to remote node via the network router. When a remote request is received at its destination node, it is merged into the local-request queue of the remote node. Similarly, the responses are split into the local-response queue and the remote-response queue depending on their requests' origins. The response network router forwards the non-local KVS responses to the remote nodes, which are later merged into the local response queues at their sources. The request and response networks are kept separate using the support for virtual networks in our network implementation [128]. With

the support of virtual networks, Bluecache can also dynamically assign a dedicated network channel for a running application, so that the network interference between applications is minized.

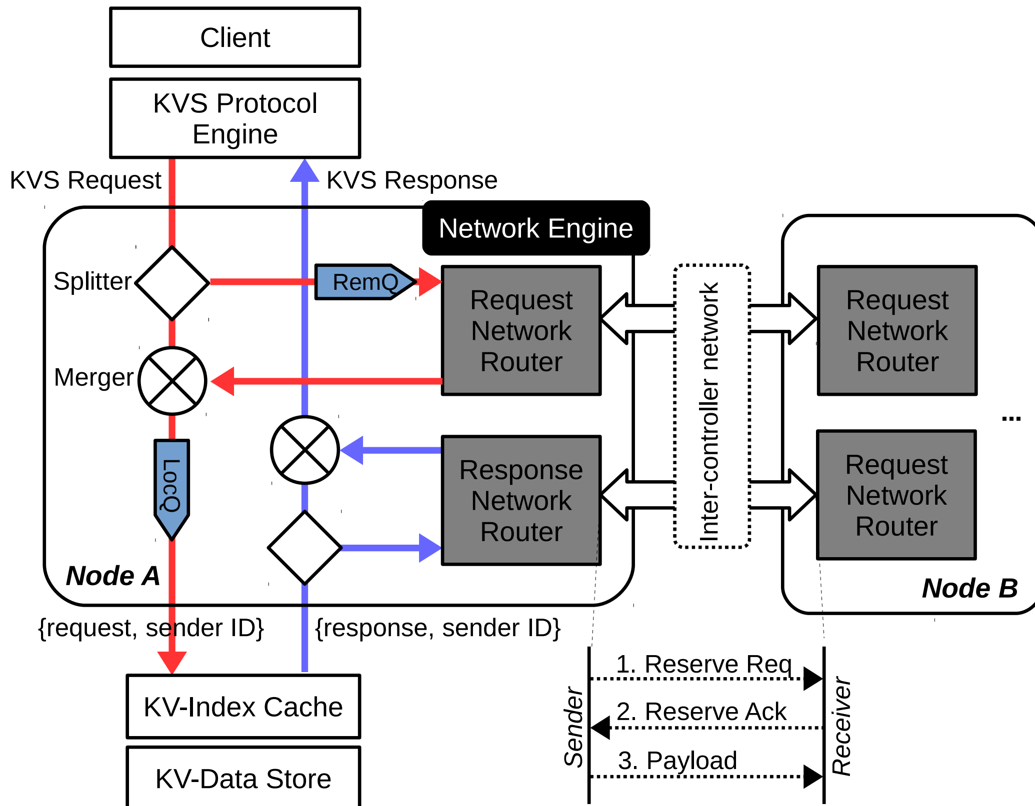


Figure 3-4: Network Engine Architecture

Our multi-FPGA platforms network is designed to be lossless and thus, uses a simple *handshake* network protocol. Before sending a payload, the sender sends a reserve request to the receiver with the size of the payload. The receiver then reserves the memory for the incoming payload, and acknowledges the sender. After receiving the acknowledgment, the sender sends the payload data to the receiver. Because of the simplicity of the protocol, our network has only a 1.5μs latency per hop.

3.3 KV-Index Cache Manager

KV-index cache stores key-value metadata such as timestamps, which has frequent updates. BlueCache keeps KV-index cache in DRAM because in-place index updates on flash would make it prohibitively inefficient and complicated. To facilitate this, BlueCache organizes the data structures of KV-index cache and KV-data store differently than DRAM-based solution (See Figure 2-3). Like Fatcache [199], abbreviated keys instead of the full keys are kept in KV-index cache. The abbreviated key is SHA-1 hash of the key, and it acts as an unique identifier for each object. The complete key is stored together with the value in the KV-data store. False positives from SHA-1 hash collision are detected when objects are retrieved from KV-data store and compared with the requested key. Such data structure organization ensures that 1) there are no flash accesses because of KV-index-cache misses; and 2) on a KV-index cache hit, a single flash access is needed to access both the key and data. This organization also quickly detects cache misses which can arise because of rare SHA-1 hash function collisions.

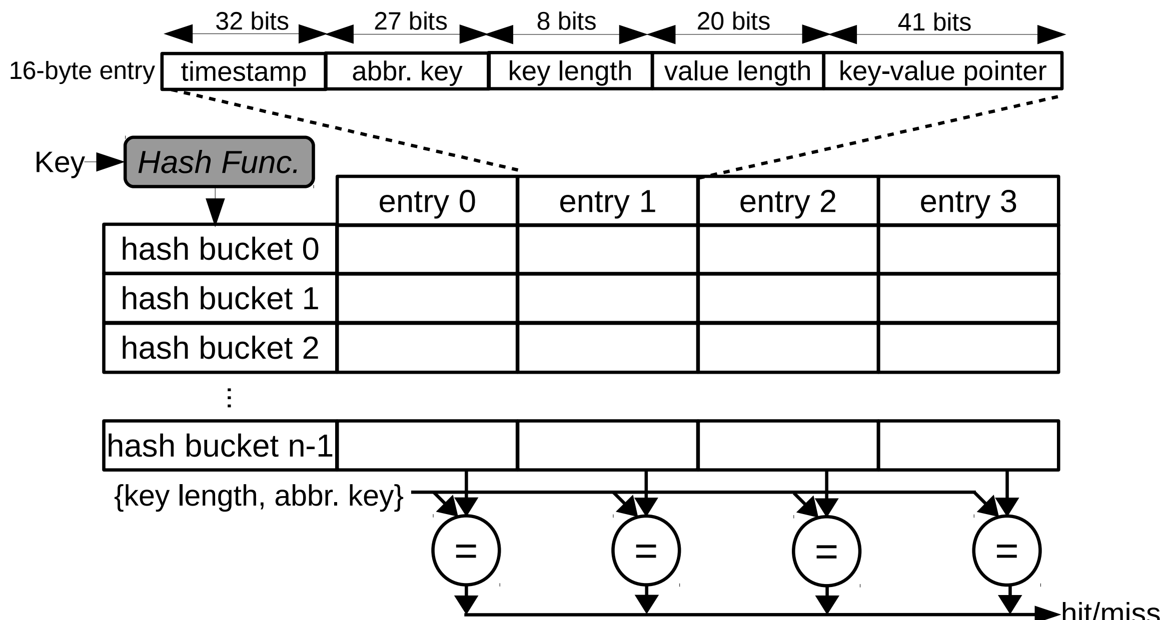


Figure 3-5: 4-way set-associative KV-index cache

The KV-index cache is organized as a *set-associative* hash table on DRAM (Figure 3-5). A key is mapped into a hash bucket by computing the *Jenkins hash function* [4], which is

also used in memcached. A hash bucket contains fixed number of 128-bit index entries, each of which contain 5 fields describing a key-value pair. A 41-bit *key-value pointer* field points to the location of the key-value pair in a 2TB address space; a 8-bit *key length* field represents the key size (up to 255B); a 20-bit *value length* field represents value size (up to 1MB); a 32-bit *timestamp* field represents the key-value pair's latest access time, which is used to evict an index entry; and a 21-bit field stores the truncated SHA-1 hash value of the key to resolve hash function collisions. The maximum key and value sizes are chosen as in memcached. If there is a need to support a larger key or value size, the index entry has to be changed by either increasing the size of the index entry, or decreasing the abbreviated key field size within an acceptable false positive hit rate.

Since it is inefficient to use linked list or rehashing to handle hash function collisions in hardware, we check all four index entries in the hash bucket *in parallel*. For a KV-index cache hit, the requested key needs to match the abbreviated key and the key length.

Because our FPGA platform returns 512-bit data per DRAM request, we assign each hash bucket to every 512-bit aligned DRAM address. In this way, each hash bucket has four index entries. Yet, such a 4-way set-associative hash table design can guarantee good performance since each hash table operation only needs one DRAM read to check keys, and one DRAM write to update timestamps and/or insert a new entry. Higher set associativity can reduce conflict misses of KV-index cache and increase performance in theory. Yet, increasing associativity requires more hardware, more DRAM requests and more clock cycles per operation, which can offset benefits of a higher hit rate.

A simple calculation can also demonstrate the rarity of false positive hits with such a simple hash table design. An 8GB 4-way set-associative hash table with 128-bit per entry $2^{33-2-4} = 2^{27}$ hash buckets. With 21 bits for abbreviated keys, the false positive for key hit on a index entry is as low as $1/(2^{27+21}) = 1/2^{48}$. This ensures that KV-index cache itself is able to filter out most misses, and that a negligible portion of misses are penalized by expensive flash reads.

When there are more than four collisions at a hash bucket, an old index entry is evicted. Since KVS workload shows strong temporal locality [48], the KV-Index Cache Manager uses LRU replacement policy to select the victim by reading their timestamps. When a

key-value pair is deleted, only its index entry is removed from KV-index cache, and the object on KV-data store is simply ignored, which can be garbage collected later.

3.4 KV-Data Store Manager

The key-value object data is stored either in the DRAM or in the flash (Figure 3-6). Key-value pairs are first written in DRAM and when it becomes full, less popular key-value pairs are evicted to the flash to make space. Each entry in the KV-data store keeps a backward pointer to the corresponding entry in the KV-index cache. When the KV data is moved from the DRAM to the flash, the corresponding KV index entry is updated. The most significant bit of key-value pair pointer in the KV-index cache indicates whether the data is in DRAM or flash.

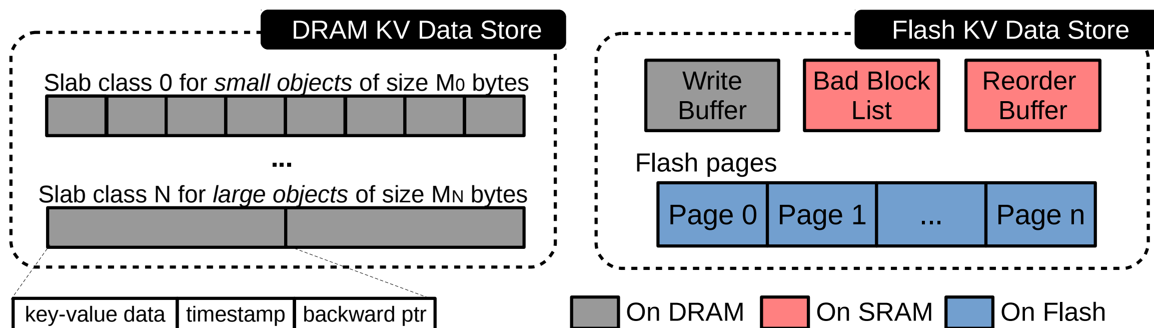


Figure 3-6: KV-data store architecture

3.4.1 DRAM KV-Data Store

The DRAM KV-data store is organized as a collection of slab classes, where each slab class stores objects of a predetermined size (Figure 3-6). A slab-structured DRAM store implementation requires simple hardware logic, and its dense format enables better RAM efficiency. The KV-data store in memcached [167] uses a similar technique in order to minimize DRAM fragmentation in software. When an object needs to be moved to the flash, the victim is determined by the entry with the oldest timestamps amongst four randomly-chosen entries in the slab. After the victim is evicted, the in-memory KV-index cache is

updated using backward pointer of the evicted object. This policy behaves like a pseudo-LRU replacement policy, which keeps hot objects in DRAM. Since the KV-Data Store shares the DRAM with the KV-index cache, and the size of DRAM KV-data store can be dynamically adjusted in respect to the size of active KV-index cache.

3.4.2 Flash KV-Data Store

The data structure on the KV flash data store is very different from the DRAM KV-data store. On flash one has to erase a page before it can be overwritten. Though flash chip allows reads and writes at the granularity of a page, it permits only block erasures, where the typical size of a block is 64 to 128 pages. Block erasure has high overhead (several milliseconds) and is performed in the background as needed. To avoid small overwrites, we use log-structured techniques [147, 148] for writing in flash. Victims from the DRAM KV-data store are collected in a separate DRAM buffer, and written to flash in 128-page chunks as a log (Figure 3-6, Figure 3-7). The KV-data manager has the flexibility to map the data pages on the flash array and instead writing a 128-page chunk on one chip, it stripes the chunk across *N* flash chips for maximum parallelism. This also ensures good wear-leveling of flash blocks. The manager also has to maintain a list of bad blocks, because flash blocks wear out with usage (Figure 3-7).

Similar to the work of Chen *et al* [71], the KV-Data Store Manager schedules reads concurrently to maximize parallelism, and consequently, the responses do not necessarily come back in order. A reorder buffer is used to assemble pages to construct the response for a request [160].

Flash KV-data store also implements a minimalist garbage collection algorithm for KV-data store. When space is needed, an old flash chunk from the beginning of the log is simply erased and overwritten. The keys corresponding to the erased data have to be deleted from the KV-index cache to produce a miss. In KVS workloads, newly written data has higher popularity [48] and by overwriting the oldest data, it is more likely that cold objects are replaced by hot ones. Consequently new objects are always written in the DRAM cache. BlueCache's simple garbage collection ensures good temporal locality.

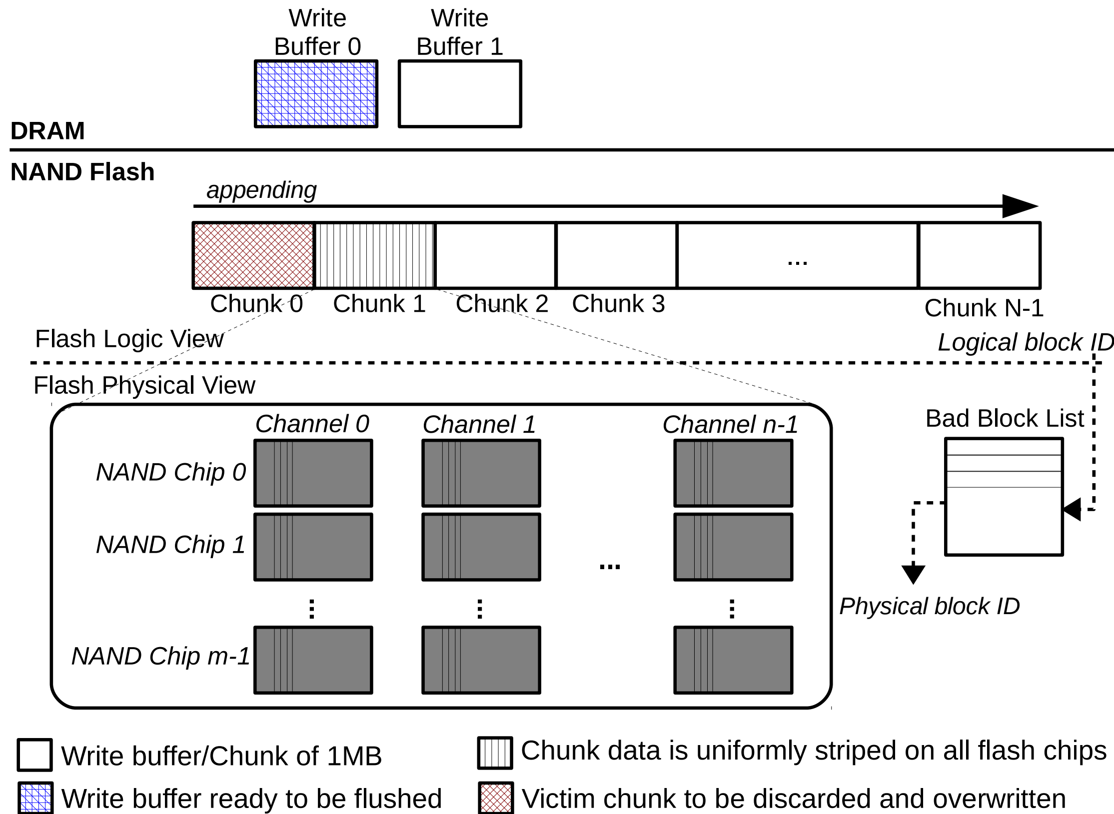


Figure 3-7: Log-structured flash store architecture

Such a KV-data store requires direct accesses to NAND flash chips, which is very difficult to implement with commercial SSDs. Manufactures often deploy a flash translation layer (FTL) inside SSDs to emulate hard drives behaviors, and this comes in the way of exploiting the full flash bandwidth [60, 171]. The BlueCache architecture eliminates FTL and uses direct management of parallel NAND flash chips to support KVS operations.

3.5 From KVS Cache to Persistent KVS

The BlueCache architecture described so far does not use the non-volatility of flash; it only exploits the capacity advantage of flash over DRAM. However, it can be easily transformed into a persistent KVS.

In order to guarantee persistency, BlueCache needs to recover from crashes due to power outage as well as other system failures. If the KV-index cache is lost then the KV-data store

essentially becomes inaccessible. Like filesystems [144, 148], we can periodically write the KV-index cache to flash for persistency. The KV-index cache checkpoints are stored in a different region of the flash than the one hold the KV data. In this way, a slightly older version of KV-index cache can be brought back in DRAM quickly. We can read the flash KV-data store, replay log from the timestamp of the KV-index cache checkpoint, and apply SET operation on the key-value pairs sequentially from the log to recover the KV-index cache. When checkpoints are being made, updates to KV-index cache should not be allowed. To allow high availability of the KVS, we could devise a small interim KV hash table, which can be merged later with KV-index cache and KV-data store after the checkpoint operation finishes.

The write policy for the DRAM data cache should depend on the requirement of data recovery by different applications. For applications that need fast retrieval of hot data, write-through policy should be chosen because key-value pairs will be immediately logged on to flash. Yet, this comes at the cost of having a smaller effective KVS size. On the other hand, write-back policy works for applications which have relaxed requirement for recovering the most recent data and benefit from larger capacity.

3.6 Architectural Innovations

We have described the hardware component of BlueCache. The following highlights the optimiztions applied on each component to maximize the overall performance of BlueCache.

1. Hardware-assisted auto-batching for the KVS request processing.

2. In-storage hardware-managed network for KVS node communication with dynamic allocation of dedicated virtual channels for different applications.

3. Hardware-optimized set-associative KV-index cache.

4. Elimination of FTL with log-structured KVS flash manager, which implements simple garbage collection and schedules out-of-order flash requests to maximize parallelism.

As discussed in Chapter 2, almost every element of BlueCache is present in some work on KVS and/or flash system. Yet our system architecture as a whole is unique. BlueCache presents novel flash-based KVS solution which has balanced architecure that can fully utilize flash's performance.

3.7 BlueCache Software Interface

In data centers, a KVS cluster is typically shared by multiple application. BlueCache provides a software interface, which allows many multi-threaded applications to share the KVS concurrently. The software interface implements a partial memcached client API consisting of three basic C++ functions: GET, SET, DELETE, as listed below. The C++ functions can be also accessed by other programming languages via their C wrappers, such as JAVA through Java Native Interface (JNI).

```
bool bluecache_set(char* key, char* value, size_t
   key_length, size_t value_length);

void bluecache_get(char* key, size_t key_length, char**
   value, size_t* value_length);

bool bluecache_delete(char* key, size_t key_length);
```

These functions provides *synchronous* interfaces. The BlueCache KVS throughput via the software interface increases as there are more concurrent accesses, and it stops scaling beyond 128 concurrent application threads (See Section 4.5).

BlueCache software interface is implemented by using three types of threads (Figure 3-8). *Application thread*s send KVS queries via the API. *Flushing thread* periodically pushes partially filled DMA segments to host interface in background if there are no incoming requests. *Response thread* handles DMA interrupts from hardware, and dispatches responses to the application threads.

Furthermore, BlueCache software interface has three data structures owned by different threads. The *request queue* buffers all KVS requests, and is shared by the application

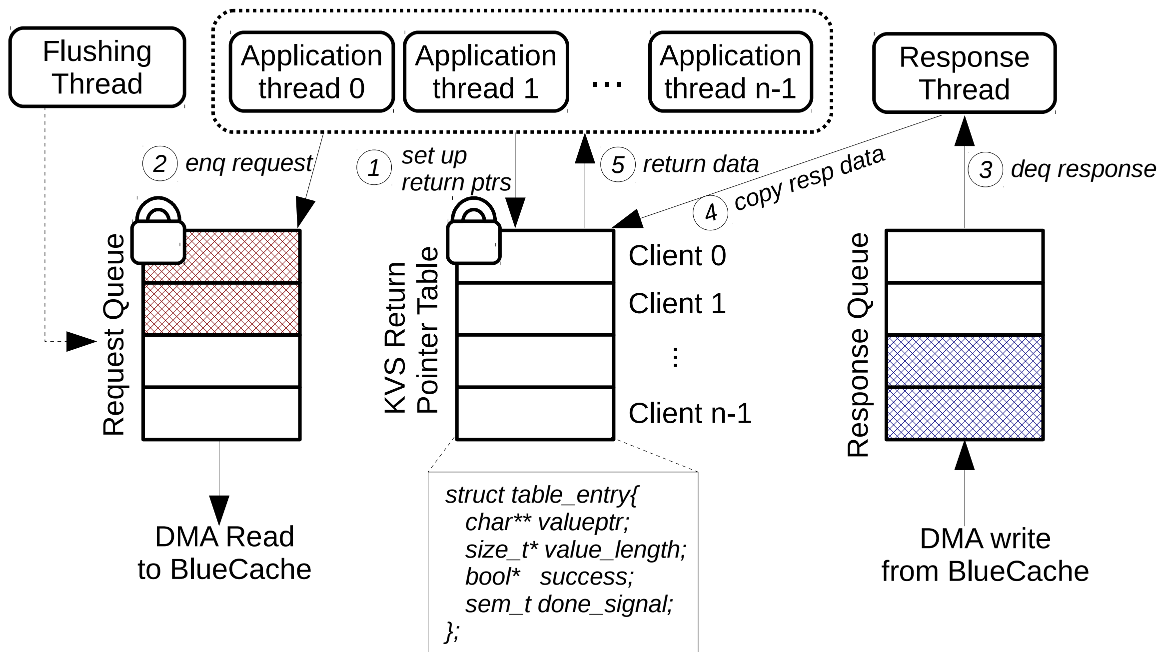


Figure 3-8: BlueCache software interface

threads and the flushing thread. The *response queue* buffers all KVS responses returned from hardware, which is solely owned by the response thread. The *KVS return pointer table* is shared by the application threads and the response threads, and maintains an array of return pointers for each clients. Once the hardware returns KVS query results, the response thread can signal the right application thread with the data. All the shared data structures are protected by mutex locks.

Figure 3-8 also illustrates the internal operations of the software interface. When an application thread queries BlueCache, it first sets up its return pointers on the KVS return pointer table. Second, the application thread push the KVS queries to the request queue, which is later send to BlueCache via DMA. The client thread then waits for the response. The response thread receives an interrupt from the hardware after BlueCache pushes KVS query results to the response queue. The response thread then dequeues the KVS response, copies response data into a byte array, and signals the application thread with the response by referring to the information on the KVS return pointer table.

Chapter 4

Implementation and Evaluation

We use Field Programmable Gate Arrays (FPGA) to implement BlueCache. In this chapter we describe BlueCache's implementation platform, hardware resource usage, and power characteristics in order. And then we will evaluate BlueCache using micro-benchmarks and an end-to-end social networkign benchmark.

4.1 BlueDBM Platform

BlueCache is built on BlueDBM Platform as described in Section 1.4. BlueCache KVS components are mapped into different BlueDBM parts. The Xeon computer nodes are used as application servers, and BlueDBM storage nodes are used as the KVS. On each BlueDBM storage node, the KV-index cache is stored in the DDR3 SODIMM, and the KV-data store is stored in the flash. The high-speed serial links are used as the BlueCache network. The Virtex-7 FPGA hosts all the hardware accelerators, which are developed in the high-level hardware description language Bluespec [12].

On the BlueDBM's Xeon servers, the software interface is developed on Ubuntu 12.04 with Linux 3.13.0 kernel. We used the Connectal [136] hardware-software codesign library which provides RPC-like interfaces and DMA over PCIe. We used a PCIe gen 1 version of Connectal, which supports 1 GB/s to and 1.6GB/s from FPGA. Two flash cards would provide a total of 2.4 GB/s bandwidth which would exceed the PCIe gen 1 speed; however, only one flash card is used due the limitation of FPGA resources to be explained in Section 4.1.1.

4.1.1 FPGA Resource Utilization

Table 4.1: Host Virtex 7 resource usage

Module Name	LUTs	Registers	RAMB36	RAMB18
KVS Protocol Engine	17128	13477	8	0
Network Engine	74968	184926	189	11
KV Data Store Manager	33454	32373	228	2
KV Index Cache Manager Table	52920	49122	0	0
Virtex-7 Total	265660 (88%)	227662 (37%)	524 (51%)	25 (1%)

Table 4.1 shows the VC707 resource usage for a single node of a four node configuration with one active flash card per node. This configuration uses most (88%) of the LUTs and half the BRAM blocks. Given the current design, the VC707 does not support a full-scale BlueCache KVS by using all the hardware resources provided by BlueDBM platform (20-node KVS node with 20TB flash capacity).

4.2 Power Consumption

Table 4.2: BlueCache power consumption vs. other KVS platforms

Platforms	Capacity *(GB)*	Power *(Watt)*	Capacity/Watt *(GB/Watt)*
FPGA with Raw Flash Memory(BlueCache)	20,000	800	25.00
FPGA with SATA SSD(memcached) [60]	272	27.2	10.00
Xeon Server(FlashStore) [85]	80	83.5	0.96
Xeon Server(optimized MICA) [153]	128	399.2	0.32
Xeon Server+GPU(Mega-KV) [216]	128	899.2	0.14

Table 4.2 compares the power consumption of BlueCache with other KVS systems. Thanks to the lower power consumption of FPGAs, one BlueCache node consumes only about 40 Watts at peak. A 20-node BlueCache cluster consumes 800 Watts and provides 20TB of key-value capacity. Compared to other top KVS platforms in literature, BlueCache has the highest capacity per watt, which is at least 25X better than x86 Xeon server platforms, and 2.5X over an FPGA-based KVS with SATA SSDs. A production system of BlueCache

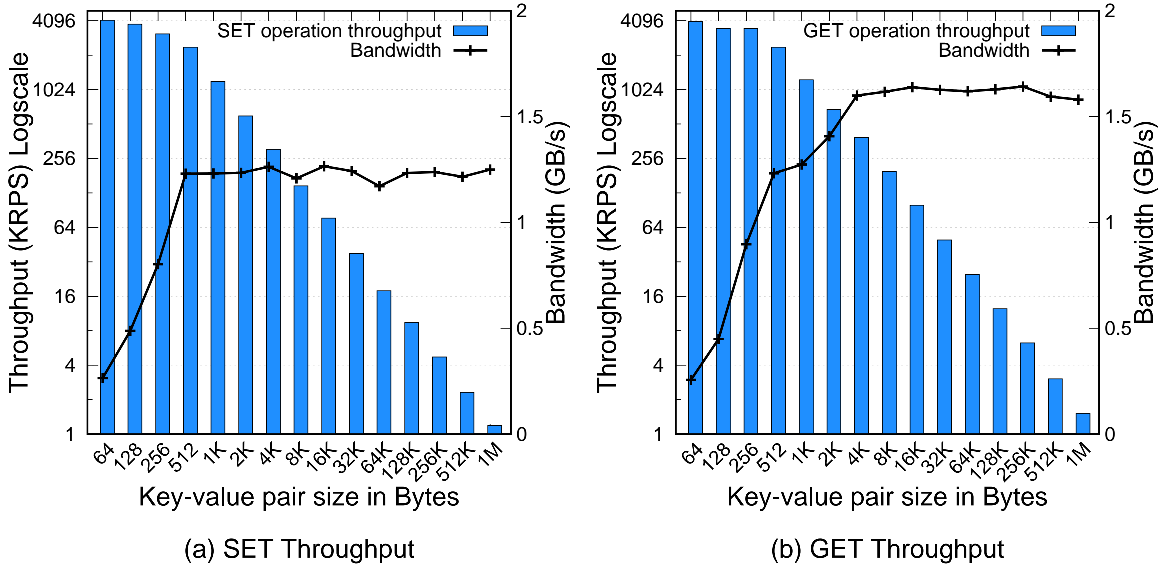


Figure 4-1: Single-node operation throughput on DRAM only

can easily support 8TB NAND flash chips *per node*, and a single node can provide 2.5M IOPs of random 8KB page read and comsumes less than 50W (1/8 of a Xeon server) [184].

4.3 Single-Node Performance

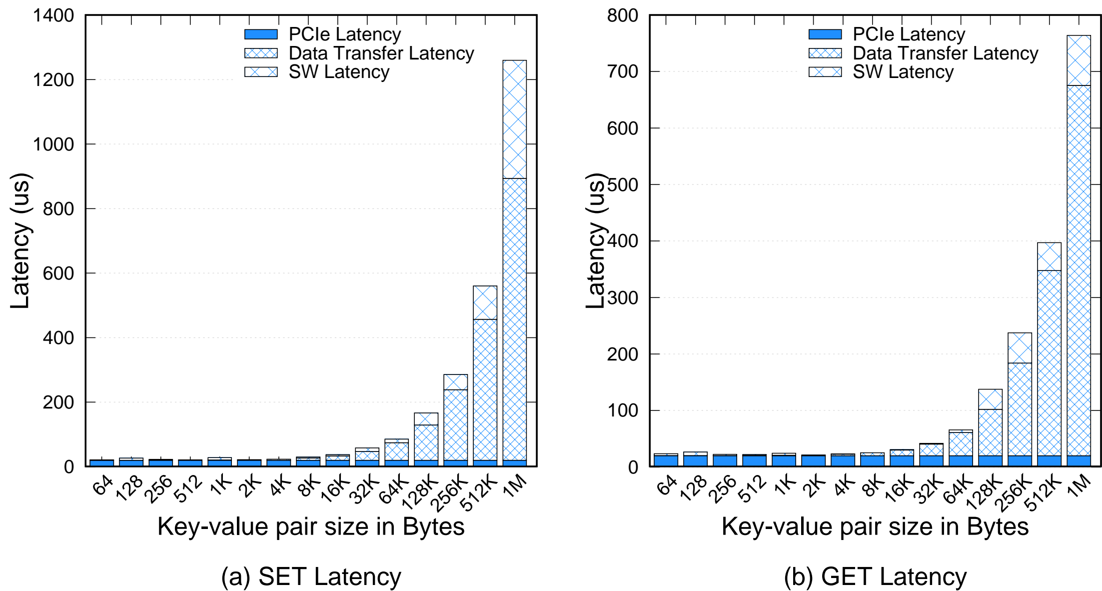
We evaluated GET and SET operation performance on a single BlueCache node. We measured both throughput and latency of the operations. Measurements are made from application servers without multi-thread software interface to test the peak performance of the BlueCache hardware.

4.3.1 DRAM-only Performance

This part evaluates the performance of a single-node BlueCache DRAM-only implementation. All key-value pairs are resident on the slab-structured DRAM store of the KV data cache. Measurements are made with key-value pairs of different sizes, keys are all 32B.

Operation Throughput

Figure 4-1 shows the throughput of a single-node DRAM-only BlueCache. DRAM-only BlueCache has peak performance of 4.14MRPS for SET operations and 4.01MRPS for GET.



(a) SET Latency (b) GET Latency

Figure 4-2: Single-node operation latency on DRAM only

This is a 10X improvement over the stock memcached running on a Xeon server which support 410 KRPS at peak.

BlueCache operation throughput is limited by different hardware components depending on the size of key-value pairs. For small key-value pairs (<512B), operation throughput is bottlenecked by random access bandwidth of the DDR3 memory. DRAMs support fast sequential accesses, but is slow for random accesses. As measured from the 1GB DDR3 DRAM, bandwidth is 12.8GB/s for sequential access vs 1.28GB/s for random access. For large key-value pairs (>512B), DRAM sequential accesses dominate but PCIe bandwidth limits the performance of BlueCache. PCIe gen1 limits transfers from the application server to BlueCache at 1.2GB/s for SETs and 1.6GB/s in the reverse direction for GETs.

Operation Latency

Operation latency consists of PCIe latency, data transfer latency and software latency. Figure 4-2 shows that operation latency of single-node DRAM-only BlueCache varies from 20μs to 1200μs. PCIe latency is constant about 20μs (10μs per direction), and it is the dominant latency source when key-value pairs are small(<8KB). Data transfer latency increases as key-value pairs become larger, and it dominates latency for key-value pairs larger than 8KB. Software latency includes processing time of PCIe interrupts, DMA requests

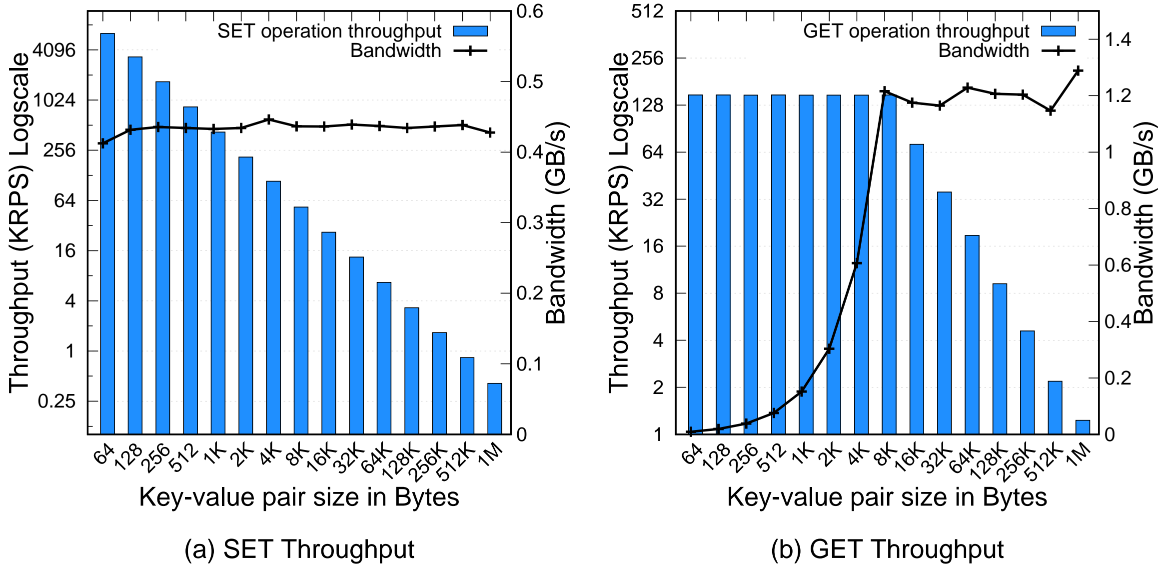


Figure 4-3: Single-node operation throughput on flash

and other software components. Software latency is small because only one interrupt is needed for key-value sizes smaller than 8KB. For bigger key-value pairs, there are more DMA bursts which requires more interrupts per KVS request, and the software overhead becomes significant.

4.3.2 Performance with Flash

This section evaluates the performance of a single-node BlueCache implementation with flash. All key-value pairs are resident on the log-structured flash store of the key-value data cache. Measurements are made with key-value pairs of different sizes, keys are all 32B.

Operation Throughput

Figure 4-3 shows the throughput of a single-node BlueCache with one flash board. Using flash, BlueCache has a peak performance of 6.45MRPS for SET operations because writes are first buffered in DRAM before writing to flash. GET operations with 64 byte key-value pairs achieve 148.49KRPS.

For SET operations, key-value pairs are buffered in DRAM and then written to flash in bulk at the 430MB/s NAND Flash write bandwidth (Figure 4-3(a)). For 64-byte key-value pairs, SET operations on flash has more bandwidth than DRAM (Figure 4-3(a) vs.

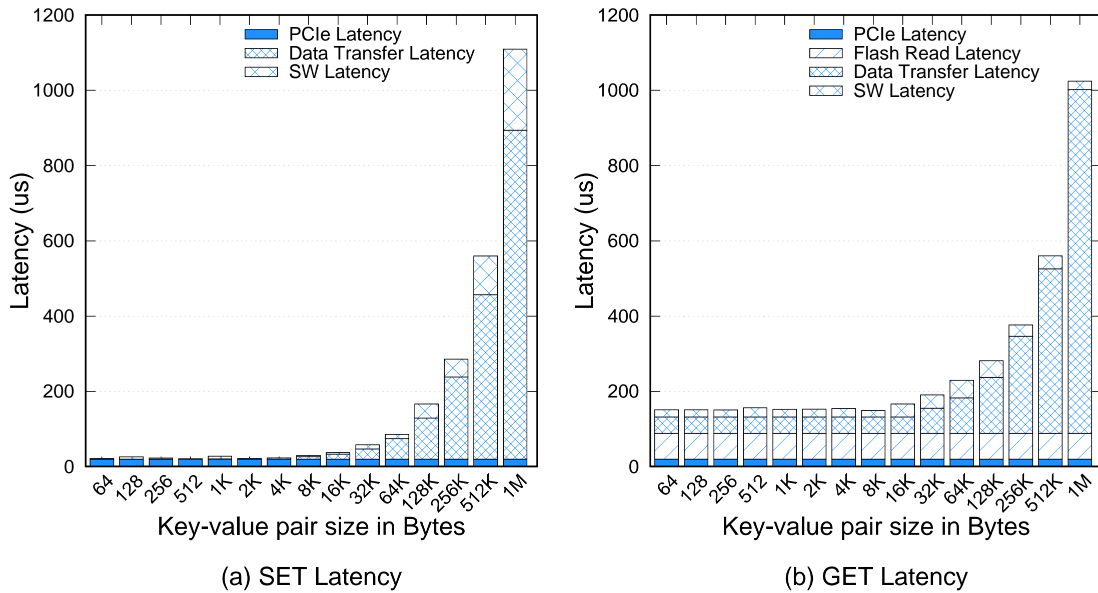


Figure 4-4: Single-node operation latency on flash

Figure 4-1(a)). The slower performance on DRAM is due to poor random accesses on the single-channel DDR3 SDRAM. For flash, performance is better because there are more sequential DRAM write pattern from the DRAM write buffer on the flash KV data store.

For GET operations, the requirement to read 8KB NAND pages limits throughput to 148K IO/s. As shown in Figure 4-3(a), for key-value pairs smaller than 8KB, only one page of flash is read and so the operation throughput is same as that of flash (~148KPS). For key-value pairs greater than 8KB, multiple pages are read from flash, and operation throughput is limited by flash read bandwidth.

Operation Latency

Figure 4-4 shows operation latency of a single-node BlueCache. SET operation latency of BlueCache using flash varies from 21μs to 1100μs, which is similar to that using DRAM only, because all SET operations are buffered in DRAM before being written to flash. GET operation latency consists of PCIe latency, flash read latency, data transfer latency, and software latency. GET operation latency varies from 150μ to 1024μs. Data transfer latency dominates, because an entire flash page needs to be fetched even for small reads.

In conclusion, raw flash device latency is the dominant latency source of BlueCache processing on flash store. And accelerating GET/SET operations on FPGA with flash adds a

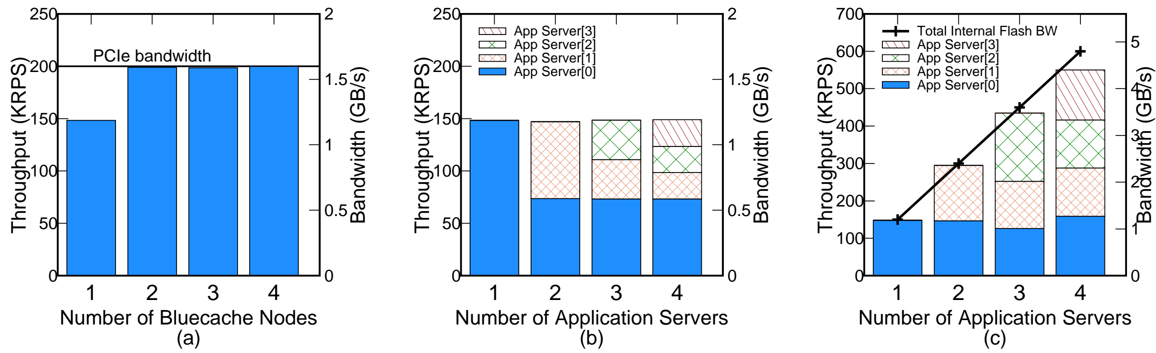


Figure 4-5: Multi-node GET operation bandwidth. (a) single app server, multiple BlueCache nodes, (b) multiple app servers, single Bluecache node, (c) multiple app servers, multiple BlueCache nodes

negligible latency to the overall key-value processing time.

4.4 Multi-Node Performance

We measured multi-node performance by chaining four BlueCache nodes together in a linear array. Each BlueCache node is attached to an application server via PCIe.

4.4.1 Operation Throughput

We measured BlueCache throughput under the following scenarios: (1) a single application server accessing multiple BlueCache nodes (Figure 4-5(a)). (2) multiple application servers accessing a single BlueCache node (Figure 4-5(b)). (3) multiple application servers accessing multiple BlueCache nodes (Figure 4-5(c)). All accesses are 40,000 random GET operations of 8KB key-value pairs on flash.

The first scenario examines the scalability of BlueCache when there is only one application server accessing BlueCache. We observed some speed-up (from 148 KPRS to 200 KRPS) by accessing multiple storage nodes in parallel (See Figure 4-5(a)), but ultimately we are bottlenecked by PCIe (current x8 Gen 1.0 at 1.6GB/s). We are currently upgrading BlueCache pipeline for PCIe Gen 2.0, which would double the bandwidth. In general, since the total throughput from multiple BlueCache servers is extremely high, a single application server with a host interface cannot consume the aggregate internal bandwidth of a BlueCache

KVS cluster.

The second scenario examines the behavior of BlueCache when there is resource contention for a BlueCache storage node by multiple application servers. Figure 4-5(b) shows that the network engine is able to maintain the peak flash performance, but favors the application server to which it is attached. In the current implementation, half of the flash bandwidth is allocated to the application server attached to a storage node.

The last scenario illustrates the aggregated bandwidth scalability of BlueCache KVS cluster, with multiple application servers accessing all KVS nodes. The line in Figure 4-5(c) shows the total maximum internal flash bandwidth of all BlueCache nodes, and the stacked bars show overall throughput achieved by all application servers. We achieved 99.4% of the maximum potential scaling for 2 nodes, 97.7% for 3 nodes, and 92.7% for 4 nodes at total of 550.16 KRPS.

4.4.2 Operation Latency

Figure 4-6 shows the average GET operation latency for 8KB key-value pairs over multiple hops of BlueCache nodes. Latency is measured from both DRAM and flash. We measured that that each hop takes ~0.5μs, therefore BlueCache handshake network protocol only takes 1.5μs per hop. When key-value pairs are on DRAM, we observed a ~2μs increase of access latency per hop for various number of node traversals, which is much smaller than overall access latency (~25μs). Access variations from other parts of BlueCache hardware are far greater than the network latency, as shown as error bars in Figure 4-6. Because accessing remote nodes are equally as fast as local nodes, the entire BlueCache KVS cluster appears as a fast local KVS storage, even though it is physically distributed among different devices.

4.5 Application Multi-Access Performance

Applications use BlueCache's software interface (See Chapter 3.7) to access BlueCache KVS cluster concurrently. Figure 4-7 shows the performance of BlueCache's software interface when there are multiple application threads sending *synchronous* GET requests. When there are a small number of application threads, BlueCache delivers nearly raw flash

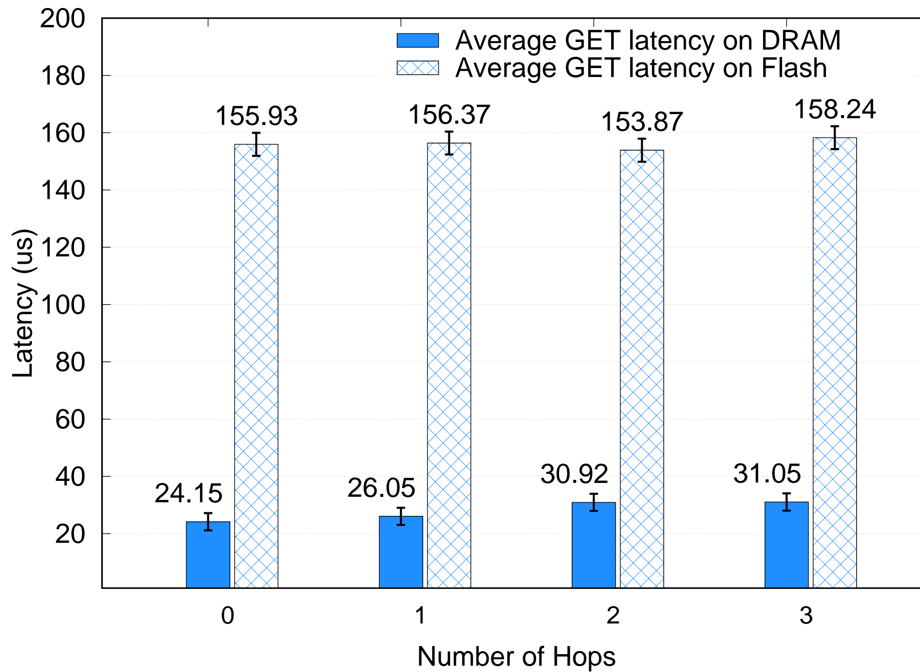


Figure 4-6: Multi-node GET operation latency on DRAM/Flash

device latency of 200μs. However, flash bandwidth is not saturated because there are not enough outstanding requests. Increasing the number of concurrent threads increases in-flight KVS requests and KVS throughput. However, if there are too many threads, latency increases to the undesirable millisecond range because of significant software overheads from context switching, locking, and request clustering. Configuring the application with 128 threads delivers 132.552 KRPS throughput at only 640.90 μs average latency on 1KB key-value pair GETs.

We also ran the same experiments on Fatcache, which is an SSD-backed software KVS implemention. We ran Fatcache on a 24-core Xeon server with one 0.5TB PCIe-based Samsung SSD and one 1Gbps Ethernet. The SSD performs like BlueCache's Flash board (130K IOPs vs 150K IOPs for 8KB page reads). Our result shows that Fatcache only provides 32KRPS throughput and 3000μs average latency. In comparison, BlueCache can fully exploit the raw flash device performance, and shows 4.18X higher throughput and 4.68X lower latency over Fatcache.

Figure 4-8 breaks down the latency of Fatcache and BlueCache requests. The actual Flash access latency is the same, but BlueCache eliminates 300μs of network latency [169],

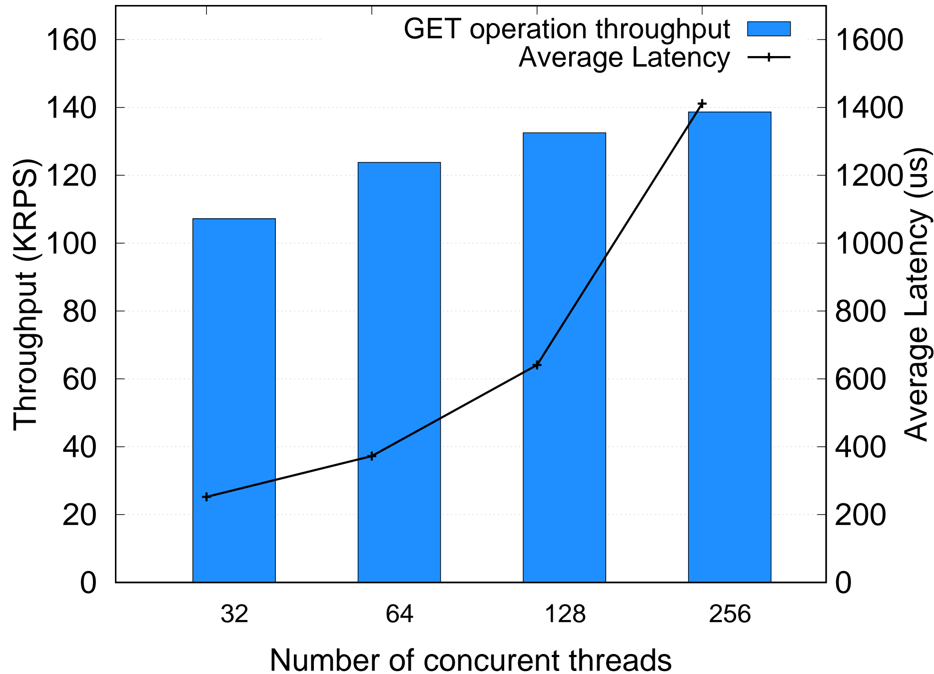


Figure 4-7: Accessing BlueCache via software interface, key-value pair sizes are 1KB

128 μs of NIC latency [169, 180], and 2000 μs of KVS software latency.

4.6 Social Networking Benchmark

We used BG [54], a social networking benchmark, to evaluate BlueCache as a data-center cache. BG consists of a back-end storing profiles for a fixed number of *users*, and a front-end multi-threaded workload simulator. Each front-end thread simulates a user performing *social actions*. Social actions consist of *View Profile, List Friends, Post Comment, Delete Comment,*

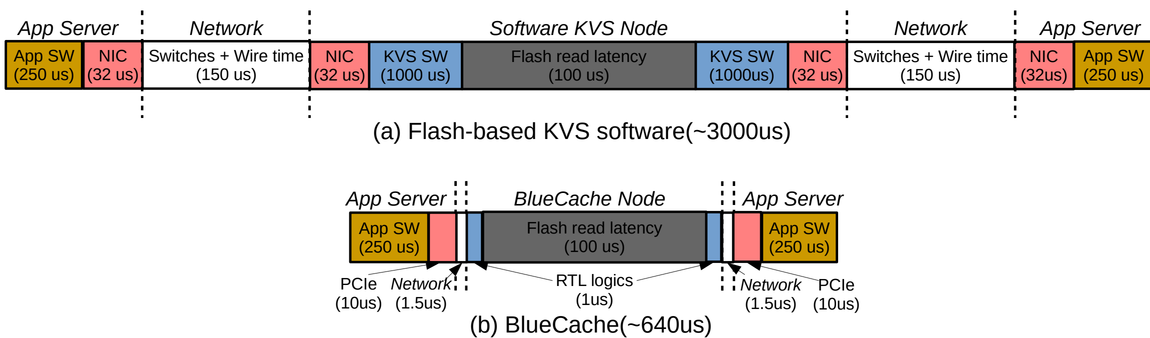


Figure 4-8: End-to-end processing latency comparison of SSD KVS software and BlueCache

and so forth.

An example of a BG backend is a MySQL database. The MySQL database persistently stores member profiles in four tables with proper primary/foreign keys and secondary indexes to enable efficient database operations. BG implements 11 social actions which can be translated to SQL statements to access the database.

BG also supports augmenting the back-end with a KVS cache. A social action type is prefixed with the member ID to form a KVS request key, the corresponding MySQL query result is stored as the value. The interactions between KVS and MySQL database are the same as were described in Section 2.2. The average size of key-value pairs of BG benchmark is 1.54KB.

We performed experiments with BG benchmark by using three different KVS systems: BlueCache, a DRAM-based KVS (stock memcached) and a flash-based KVS (Fatcache [199]). In the next two sections we describe the experiment setup and three experiments to evaluate BlueCache.

In our experiment configurations, the back-end data store of BG is a cache-augmented MySQL database. The MySQL database persistently stores member profiles in four tables with proper primary/foreign keys and secondary indexes to enable efficient database operations. BG implements 11 social actions which can be translated to SQL statements to access the database. A particular key-value store of interest augments the MySQL database as the cache. BG uses social action type with the member ID as the key, and the corresponding social action results as the value. The average size of key-value pairs of BG benchmark is 1.54KB, with maximum size of 4.6KB. For read-only social actions, BG consults the MySQL database only if results fail being fetched from the key-value cache. For social actions which update the data store, it deletes relevant key-value pairs from the cache and updates the database, to make the cache coherent with the persistent storage.

The front-end workload simulator of BG is a multi-threaded java program making requests to the back-end data store. Zipfian is used to simulate the accessing distribution by active users. BG allows us to adjust parameters such as number of active users, and mixture of different type of social actions, to examine different behaviors of back-end stores.

4.6.1 Experiment setup

MySQL Server runs a MySQL database containing persistent user data of BG benchmark. It is a single dedicated machine which has two Intel Xeon E5-2665 CPUs (32 logical cores, 2.4GHz), 64GB DRAM, 3x 0.5TB M.2 PCIe SSDs in RAID-0 (~3GB/s bandwidth) and a 1Gbps Ethernet adapter. The database is pre-populated with member profiles of *20 millions users*, which is ~600GB of data. The database is configured with a 40GB InnoDB buffer pool.

Application Server runs BG's front-end workload simulator on a single machine which has two Intel Xeon X5670 CPUs (24 logic cores, 2.93GHz), and 48GB DRAM. BG's front-end multi-threaded workload simulator supports a maximum of 6 million active users on this server, which is about 20GB of working set. The front end has a zipfian distribution with mean value of 0.27, to mimic the skew nature of the workload.

Key-Value Store caches MySQL database query results. We experimented with three KVS systems to examine behaviors of different KVSs as data-center caching solutions.

System A, Software DRAM-based KVS: System A uses stock memcached as a in-memory key-value cache, and uses 48 application threads to maximize throughput of memcached.

System B, Hardware flash-based KVS: System B uses a single BlueCache node as a key-value cache. The BlueCache node is attached to the client server via PCIe. System B uses 128 application threads to maximize throughput of BlueCache.

System C, Software flash-based KVS: System C uses Fatcache, a software implementation of memcached on commodity SSD. System C uses 48 application threads to maximize throughput of Fatcache.

Table 4.3 compares the characteristics of storage medium used by three different KVS systems.

Table 4.3: KVS storage technology comparison

KVS systems	Storage Media	Capacity	Bandwidth
memcached	DDR3 SDRAM DIMMs	15GB	64GB/s
BlueCache	NAND flash chips	0.5TB	1.2GB/s or 150K IOPs
FatCache	Samsung m.2 PCIe SSD	0.5TB	1GB/s or 130K IOPs

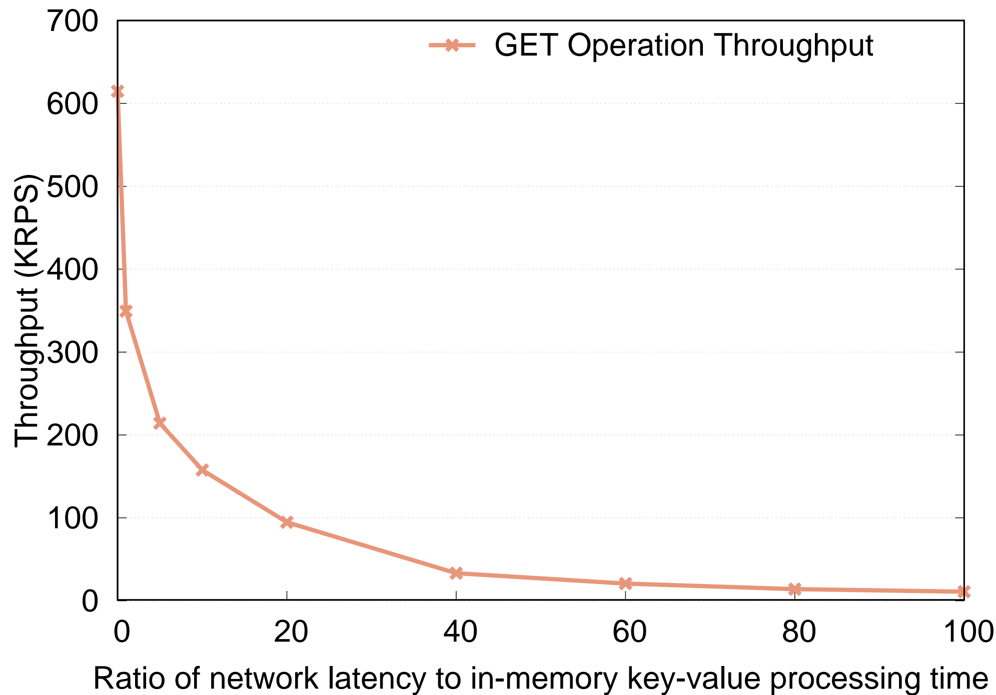


Figure 4-9: Throughput of memcached for relative network latencies

We also found that running the benchmark frontend and memcached on the same server has higher throughput than running them on separate machines. The network connection speed between the application server and the KVS plays a critical role in the overall system performance. Figure 4-9 shows that the throughput of stock memcached decreases exponentially when more of the processing time is spent on network compared to actual in-memory KVS access. We measured peak throughput of stock memcached via 1Gbps Ethernet, which is about 113 KRPS and is lower than the peak throughput of BlueCache. The 1Gbps Ethernet can limit the performance of System A and C, while BlueCache's fast network is not a bottleneck for System B. When memcached and the application runs on the same server, the throughput is 423KRPS, and the server utilizes <50% of CPU cycles at peak, which means sharing CPU cycles was not a bottleneck in such a set-up. For our experiments, we eliminated the 1Gbps network bottleneck and deployed both System A and C on the same physical machine of the application server, to have a fair comparison with System B.

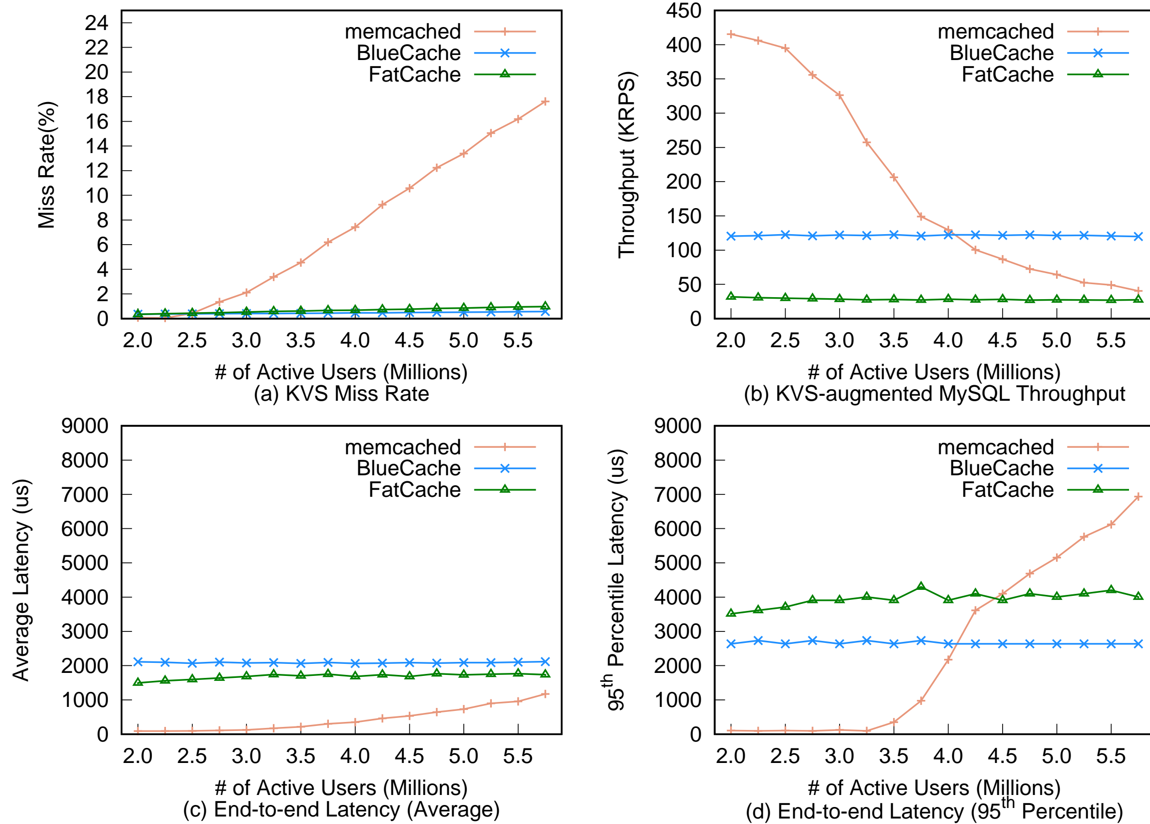


Figure 4-10: Performance of BlueCache and other KVS systems as augmentations to a MySQL database, which stores simulated data of a social network generated by BG benchmark. Evaluations are made with various numbers of active social network users.

4.6.2 Experiments

We ran BG benchmark with our experimental setup to evaluate the characteristics of BlueCache and other KVSs as data-center caches. Misses from KVS are penalized by reading the slower backend to refill the KVS cache, thus KVS miss rate is an important metric to evaluate the efficacy of the cache. There are two types of misses for KVS caches. First type happens when KVS capacity is limited and can not hold the entire working set of the application. We call them *capacity misses*. The second type of KVS misses are caused by updates to the backend. When new data is written, the application DELETEs the relevant key-value pairs in KVS to make KVS coherent with the new data in the backend. In this case, the next corresponding GET request will return a miss. We call such misses *coherence misses*.

We ran three experiments to examine how KVS misses can effect the overall performance of different KVS cache solutions.

- *Experiment 1* evaluates the benefits of a slow and large KVS cache over a fast and small one. Like typical real use cases [48, 62], the front-end application in this experiment has a very low update rate (0.1%), thus coherence misses are rare. DRAM-based solution (memcached) is faster than flash-based solution (BlueCache), but its superiority can quickly diminish as the former suffers from more capacity misses. We will show the cross point where BlueCache overtakes memcached. We will also show the superiority of a hardware-accelerated flashed-based KVS solution over a software implementation (BlueCache vs. Fatcache).

- *Experiment 2* examines the behavior of BlueCache and memcached when they have the same capacity. In such cases, both KVS solutions will experience capacity misses, and we will show the performance drop as capacity misses increase for both systems.

- *Experiment 3* examines the behavior of BlueCache and memcached when the application has more updates to the backend. In such cases, both KVS solutions will experience coherence misses, and we will show the performance drop as coherence misses increase for both systems.

Experiment 1 Results

Experiment 1 shows that BlueCache can sustain more request rate than memcached, when the latter has the more than **7.4%** misses (See Figure 4-10(a)(b)). With no capacity misses, memcached sustains 3.8X higher request rate (415KRPS) than BlueCache (123KRPS), and 12.9X higher than Fatcache (32KRPS). As the number of active users increases, the capacity of memcached is exceeded and its miss rate increases. Increasing miss rate degrades throughput of the overall system, since each memcached miss requires MySQL accesses. BlueCache and memcached meet at the cross point at 7.4% miss rate. Fatcache, on the other hand, is 4.18X slower than memcached, and its Fatcache throughput does not exceed that of

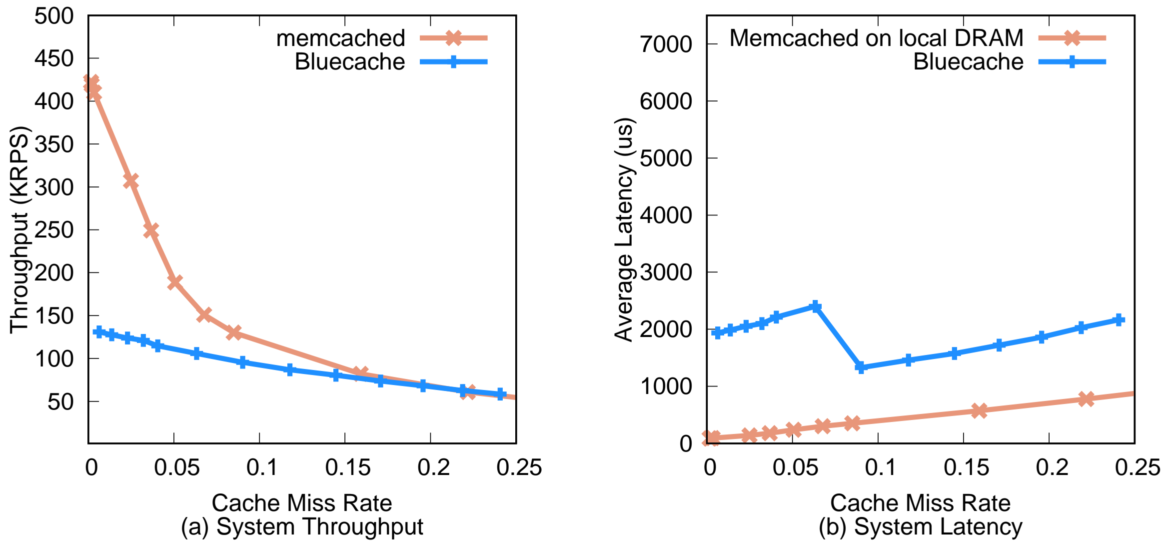


Figure 4-11: BG performance for different capacity miss rates, *both memcached and BlueCache have the same capacity of 15GB*

memcached until the number of active users exceeds 5.75 million and memcached's miss rate exceeds 18%.

Similar trend can also be found in the end-to-end application latency of BG benchmark for different KVS solutions (Figure 4-10(c)(d)). When there are no capacity misses, memcached provides an average latency of 100μs that is 20X faster than BlueCache.When memcached suffers more than 10% misses, BlueCache shows much better 95$^{\text{th}}$ percentile latency (2600μs) than memcached (3600μs-6900μs), because of miss penalties by MySQL. On the other hand, Fatcache shows about similar average latency with BlueCache (1700μs vs. 2100μs), but it has much larger variations in the latency profile and has 1.5X shorter 95$^{\text{th}}$ percentile latency than BlueCache (4000μs vs. 2600μs).

Experiment 2 Results

Experiment 2 examines the behavior of capacity cache misses of BlueCache and memcached. In this experiment, we configured the BlueCache to the same size of as memcached of 15GB, and made the benchmark to issue *read-only* requests. Thus, all misses are capacity misses. Figure 4-11(a) shows that throughput of both KVSs decreases as miss rate increases, with memcached dropping at a faster pace. Beyond 16% miss rate, both KVSs merge at

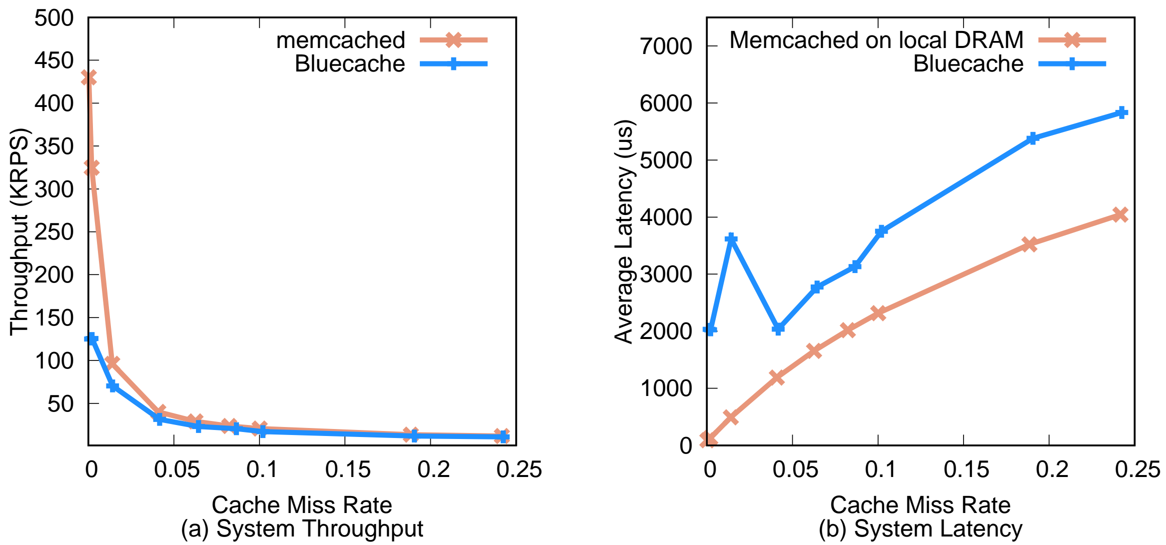


Figure 4-12: BG performance for different coherence miss rates

the same throughput when the backend becomes the bottleneck. Figure 4-11(b) shows that latency also increases as miss rate increases, but memcached delivers shorter latency than BlueCache in general. On Figure 4-11(b), there is a breakpoint of the latency for BlueCache, because of change of thread counts of the benchmark. The benchmark changes the thread count from 128 to 64 at the breakpoint, in order to reduce request congestion and lower latency while maintaining maximum throughput. In an ideal case, the benchmark would dynamically adjust the request admission rate, so that the curve in Figure 4-11(b) would be smooth.

Experiment 3 Results

Experiment 3 examines the behavior of coherence cache misses of BlueCache and memcached. In this experiment, we vary read/write ratios of the benchmark, to control the coherence cache miss rate. Similar to Experiment 1, the throughput of both KVSs decreases as miss rate increases, with memcached dropping at a faster pace (Figure 4-12(a)). And at merely 6% miss rate, both KVSs merge at the same throughput when the backend becomes the bottleneck. Figure 4-12(b) shows that latency also increases as miss rate increases, but memcached delivers shorter latency than BlueCache in general. Similar to Experiment 2, the break point of latency for BlueCache is due to change of request thread count to reduce

request congestion from the applications.

4.7 Part I Conclusion

We have presented BlueCache, a fast distributed flash-based key value store appliance that uses near-storage KVS-aware flash management and integrated network as an alternative to the DRAM-based software solutions at much lower cost and power. A rack-sized BlueCache mini-cluster is likely to be an order of magnitude cheaper than an in-memory KVS cloud with enough DRAM to accommodate 10~20TB of data. We have demonstrated the performance benefits of BlueCache over other flash-based key value store software without KVS-aware flash management. We have demonstrated the scalability of BlueCache by using the fast integrated network.

BlueCache has a clear advantage over a DRAM-based solution, if an application can tolerate milliseconds of tail-latency, and has large size response associated with each user-query. In data centers, many applications, such as video streaming, can tolerate tail-latency of several milliseconds [24]. When average cached object size of the workload is big, say greater than 8KB, the performance-per-watt of Bluecache is almost certain to beat DRAM-based system, because large sequential reads to flash drive amortizes penalty of page read amplification, and results similar performance to DRAM-based system with superior power efficiency. In addition, when a workload has large average value size associated with its queries, it can lead to extra pressure on KVS systems for more caching capacity. Under such a scenario, Bluecache can provide orders-of-magnitude larger cache capacity at a much lower cost than DRAM-based solutions.

When a workload's KV sizes are small, BlueCache can still win over DRAM-based systems under certain conditions, especially when the overall system suffers from cache capacity misses. We have shown that the performance of a system which relies data being resident in in-memory KVS, drops rapidly even if a small portion of data has to be stored in the secondary storage. With more that 7.4% misses from in-memory KVS, BlueCache is superior solution in data centers than a DRAM-based KVS, with more affordable and much larger storage capacity.

All of the source codes of the work are available to the public under the MIT license. Please refer to the git repositories: `https://github.com/xushuotao/bluecache.git` and `https://github.com/sangwoojun/bluedbm.git`.

4.8 A Retrospective on Hardware-acceleration near Flash for Latency-sensitive Workloads

BlueCache implements the entire flash-based KVS stack in hardware, and showed in 2017 that such an hardware-centric approach could achieve 25× power-efficiency improvement over general-purpose solutions. By integrating compute, network and storage layers tightly in hardware, BlueCache could process queries at hundreds of microseconds latency at the I/O device limits, without incurring software overhead.

Since 2017, when BlueCache first came in the public view, general-purpose machines have made great strides in providing low-latency solutions. Intel integrated DDIO technology in their new Xeon cores, which enables direct I/O traffics to and from last-level cache and allows x86 cores to process I/O packets with a much lower latency [116]. Industry also has open-sourced many software development packages such as DPDK [117] and SPDK [37], which reduces or bypasses the high operating system overhead in accessing I/O devices. Because of such advancements, a single core can process near a million of remote storage I/O requests per second with several hundreds of microseconds latency [137]. Moreover, recent academic projects have shown that, by aggressively optimizing software scheduling, latency-sensitive applications running on general-purpose machines can achieve μs-level tail-latency, in spite of unpredictable interference from other co-hosted workloads [99, 137, 168].

Many recent software systems [52, 93, 150, 151] have also used latest NVM technology, such as 3D Xpoint memory technology [18], to overcome the high-cost of in-memory computing like BlueCache. In 2018 Facebook used Intel Optane drive as the secondary cache of DRAM in this production persistent KVS software, which greatly reduced the amount of DRAM required without compromising applications' throughput, latency and storage efficiency [93].

Although software solutions have shown great improvement in enabling low-latency systems [99, 137, 168], and lowered the cost of in-memory computing by replacing DRAM with NVM in some cases [93], it does not make BlueCache obsolete as a hardware-centric solution for low-latency workloads in data-centers.

Advantages of hardware acceleration over general-purpose solutions

First, hardware accelerators are still 10× better than general-purpose solutions in terms of *pure performance*. To this date, the state-of-the-art in-memory KVS solution, KV-direct [152], uses FPGAs to accelerate networking and KV-processing like BlueCache. It is able to provide 1.22 billions KV requests per second in a single server chassis, which is an order-of-magnitude performance improvement over x86 machines [152]. Most importantly, KV-direct can keep the tail latency below 10μs [152], which is still difficult for general-purpose machines to achieve today. BlueCache has more than 100μs latency because of NAND flash access is 10×-100× longer than DRAM. Therefore, the latest state-of-the-art general-purpose solutions can have sufficient latency headroom to compete with BlueCache [99, 137]. However, low-latency NVMs are on the horizon: Intel ReRAM-based NVMs [19] are now available provide 10μs latency, which could be used to upgrade BlueCache to lower its access latency similar to that of KV-direct.

Second, hardware accelerators are orders-of-magnitude more *power-efficient* than general-purpose solutions because of specialization. KV-direct is able to improve the power consumption of in-memory KVS by 3× over CPU-base solutions [152]. And BlueCache is able to improve at least 25× better power per GB than software systems running on Xeon servers. Typically server-class machines use highly complex out-of-order (OoO) superscalar processors with tens to hundreds of megabytes of LLC to compute data-center workloads. Complex OoO superscalar cores could result in low efficiency in processing KVS operations, and incur power consumption overhead. A past study shows that traditional super-scalar general-purpose core pipeline can be grossly underutilized in performing the required KVS computations (networking, hashing and accessing key-value pairs) [156]. As discussed in Section 2.3.1, the last level data cache, which typically takes half of the size of processor area, could also be inefficient and waste energy for KVS cache systems, because large

hash-table-based KVS often exhibit poor locality due to random accesses to a large data-set.

One could argue that high-performance hardware acceleration of KVS operations with billions of requests per second and tens of μs latency could be an overkill for latency-sensitive applications today and that software solutions are more than adequate. Meanwhile, general-purpose machines can co-host many applications with intelligent scheduling algorithms to maximize the utilization of the power-hungry processor without sacrificing latency-sensitive applications [99]. However, there are still strong economic incentives to use hardware accelerators to save energy and burn less cores in the public clouds. For example, Microsoft Azure cloud uses FPGAs to accelerate many data-center workload, which saves CPU time/energy spent on many applications, such as networking, machine learning, search ranking and more [67, 179]. In addition, public cloud can also sell saved cores to cloud end-users and make significant business profits [189].

Limitations of hardware acceleration flash-based KVS

Although hardware acceleration of flash-based KVS can offer 10×-100× higher performance-per-watt than general-purpose solutions, BlueCache still suffers from some limitations compared to software solutions.

Limited Algorithm Complexity: In principle one can write any algorithm as a hardware accelerator, but designing a highly complex end-to-end system entirely as a hardware accelerator can lead to overhead in chip area, difficulty in design verification and etc. In fact, hardware accelerators typically strive for simplicity in order to yield low chip area and high clock frequency. For example, in BlueCache we designed highly efficient circuits, such as way-associative hash-table, circular logging and customized lossless network protocol, to make sure BlueCache a practical end-to-end design for a high-performance FPGA implementation. If more KVS features, such as range queries are needed, a more complex KVS design (*e.g.* log-structured merge tree) is required, which makes a hardware accelerator implementation infeasible. In fact, many data-structures, such as trees, are efficient to run on general-purpose machines, because they fit well in the processor memory-hierarchy. In such scenarios, we can use a hybrid approach to accelerate the commonly-used operations in hardware accelerators, such as networking, bloom-filtering, compression/decompression,

while leaving the uncommonly used complex component in software, such as garbage collection.

Highly Specialized Storage Stack: In practice, flash-based KVS systems use the storage stack to access commercial storage drives. The storage stack includes filesystem, page-cache, RAID support, FTL and more, and provide a clean and convenient programming interface for user-space applications. The legacy storage stack also takes care of many challenges that flash-based systems face. For example, page-cache mitigates read-amplification of flash storage by merging small reads to pages cached in DRAM. And FTL takes care of necessary flash management, such as wear-leveling, garbage collection, and etc. Instead of accessing flash via legacy storage stack, BlueCache uses a customized flash card with raw flash chips [160] and manages NAND flash storage with in a highly specialized way: BlueCache implemented the entire storage stack in hardware, because storage stack may incur additional overhead in accessing flash. For example, FTL is often provided as a black-box inside SSD. Latency-sensitive applications typically have no control over when garbage collection would occur, and maybe be interfered by unexpected flash-I/O spikes triggered by the FTL. Many recent works in user-space storage stack have optimized flash I/O storage stack across layers, which expose more information to applications to utilize fast I/O devices efficiently [137, 144, 148]. In light of advancements in storage stack, a future version of BlueCache could be designed as an accelerator integrated in the NIC on a general-purpose machine, which would also host multiple commercial NVM drives. The hardware accelerators on the NIC can process KV requests partially, including networking, hashing and KV index access. And later host software can take over partially-processed KV-requests, access the storage via user-space storage I/O stack and pass the results back to the accelerator. In this way, future BlueCache designs can take advantages of both the convenience of fast and efficient software storage stack and the lower-power high-speed processing by hardware accelerators.

Part II

In-storage Analytic-Query Accelerator for SQL Analytics

Chapter 5

Part II Introduction and Background

5.1 Overview and Motivation

Multi-terabyte/petabyte datasets are now commonplace for analytic workloads. In many cases, the data is stored in relational format on hard drives and analyzed by SQL database software such as Presto [26], Vertical [11], and MonetDB [38]. To process an analytic query, the database software brings the input data on demand from hard drives to DRAM, and then uses powerful CPUs to compute with this data. In a data warehousing architecture in the cloud, application servers and storage servers are separate and connected via a network [7]. Application servers initiate analytical queries, fetch data from the central storage and then process it. Such a "disaggregated" architecture is popular in the cloud because customers can scale the application servers and storage servers independently. For fast query responses, analytical software typically requires application servers to have sufficient hardware threads (i.e., virtual cores) and DRAM to hold the input data to overcome the long latency, large access granularity and limited bandwidth of central storage accesses. In April 2020 the largest storage-optimized Amazon EC2 server (i3.metal) can accommodate 8 1.9TB SSDs, and is equipped with 72 virtual cores and 512GB DRAM. Such large processing power and DRAM are needed to be able to fully exploit the high-bandwidth of SSDs. Storage throughput, because of advances in flash technology, has improved by 13X in the past decade [182], and has greatly outpaced CPUs ability to process data in memory [95,165]. As denser and faster storage devices become available in the future, it will become increasingly

difficult for storage servers to provide sufficient CPUs and DRAM to have a cost-effective balanced system.

An alternative solution is to push part of query processing to the storage to eliminate unnecessary data movement. Such a solution has been deployed in several commercial systems, for example, Oracle Exadata Server [105], IBM Netezza Machine [191], and IDM [200]. One of the most recent systems is Amazon Web Services (AWS)'s "S3 Select" feature, which pushes filter operation to the shared cloud storage service, and can get up to 4X performance benefits for these operations [28]. In this thesis we propose an in-storage Analytic QUery Offloading MAchiNe (AQUOMAN), which pushes this idea of "off-loading" query processing to storage much more aggressively (See Figure 5-1).

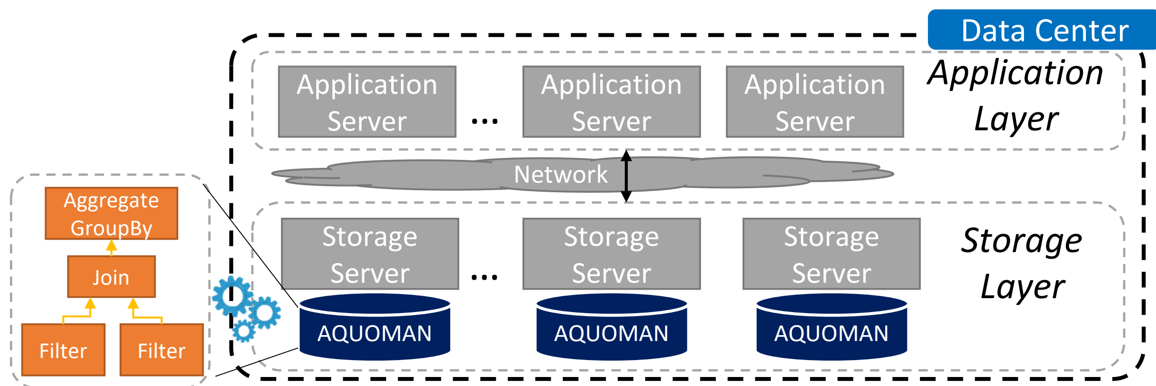


Figure 5-1: Using AQUOMAN for SQL analytics

AQUOMAN's programming model is based on a sequence of *Table Tasks*, each of which applies a static dataflow graph of SQL operators on an input table in a streaming manner to produce an output table. It takes inputs from flash, in a file format used by column-oriented database like MonetDB, because it is better suited for analytic workloads. Given the SQL query execution plan - a tree of SQL operators - we identify the subtrees that can be directly translated into *Table Tasks*. By employing a streaming model, AQUOMAN significantly reduces the DRAM requirement for intermediate tables.

We want to keep the memory in AQUOMAN to be small enough, say 16GB per 1TB-SSD, so that AQUOMAN can be *embedded* in an SSD. This prevents us from fully off-loading some queries, for example, a *multi-way join*, whose intermediate tables exceed the DRAM capacity of AQUOMAN. Despite these limitations (Section 8.5), AQUOMAN can profitably

execute the majority of queries in the TPC-H benchmark suite on a 1TB dataset, giving us an opportunity to reduce both number of hardware threads and DRAM usage in the host. We will show, using TPC-H benchmarks on 1TB data sets, that a single instance of 1TB AQUOMAN disk, on average, can free up 70% CPU cycles and reduce DRAM usage by 60%. Thus, replacing standard SSDs with AQUOMAN SSDs in database systems is a sound economic proposition.

One way to visualize this saving is to think that if we run queries sequentially and ignore inter-query page cache reuse, MonetDB running on a 4-core, 16GB-DRAM machine with AQUOMAN augmented SSDs performs, on average, as well as a MonetDB running on a 32-core, 128GB-DRAM machine with standard SSDs.

5.2 Background: Database Accelertors

Accel. Type	Related Work	Impl.	Supported SQL operator	Evaluated TPC-H Queries	Data Sz. (GB)
In-memory	Q100 [204, 205] MasterOfNone [162]	ASIC	All	All 22 queries w/o regular expression	0.01
In-storage	SmartSSD [89]	ARMs	*Filter* *Aggregate Group-By*	Q6, 2 custom queries	100
	Summarizer [140]		*Filter*	Q1,6,14, a custom query	0.1
	Biscuit [106] YourSQL [120]	ARMs& ASIC	*Filter*	All (8 is partially offloaded)	100
	Ibex [203]	FPGA	*Filter* *Aggregate Group-By*	Q13, 6, custom queries	10
	Insider [182]		*Filter*	A Custom Query	60
	FCAccel [202]		*Filter* *Aggregate Group-By* *Arithmetic*	Q1,6, and a custom query	100
	AQUOMAN	**FPGA**	**All**	**All** (14 are fully offloaded)	**1000**

Table 5.1: Representative near-data SQL accelerators

There is a long history of attempts to use specialized hardware to accelerate database query processing [63, 86, 87] but it has never caught on. One reason is that the dramatic increase in processing power and DRAM capacity of commodity hardware over the last four decades has reduced the incentive to use special hardware. However, with the rise of

specialized hardware in datacenters [67,123,179,189] and the increase of storage throughput in the last decade [182], there is a resurgence of interest in accelerating *analytical workload* using FPGAs or ASICs.

Table 5.1 gives a summary of the recent work to accelerate database operations near storage. The first family of In-Storage Processing (ISP) architectures leverages the existing ARM cores of the SSD controller to offload simple tasks like filtering [89, 140]. However, the embedded cores in the SSD controller can be 100X slower than x86 cores [197], and offloaded programs can suffer 10X slowdowns [78]. Biscuit [106] and YourSQL [120] use embedded processors in conjunction with a pattern-matching ASIC to offload filtering. They showed offloading filtering is profitable only when the selectivity is sufficiently high.

Another class of ISP (e.g. Insider [182]) uses FPGAs to add processing power to SSDs for a variety of applications to saturate large internal disk bandwidths. One example application of Insider is offloading database filtering, and it provides performance benefits similar to the one provided by a high-end ARM-based solution. Ibex [203] and FCAccel [202] use FPGAs to offload more SQL operators, such as *Aggregate Group-By*, but do not provide a plan to offload a join, which is one of the most dominant operators in analytic queries such as TPC-H.

Unlike existing ISP approaches, AQUOMAN offloads all major types of SQL operators in storage, including the computation-intensive join. AQUOMAN, given an SQL query plan, regroups SQL operators as *Table Task*s, which is the programming model for AQUOMAN's streaming architecture. This enables a transparent integration of ISP with the existing DBMS software. It is also important to note that previous database accelerator research has used much smaller data sets (10MB to 100GB [79, 89, 106, 120, 140, 202, 204]) for evaluation (Table 5.1) and has not addressed the issue of computing with large dataset. In the rest of this section we provide a more detailed discussion of the related work.

5.2.1 In-storage big data analytics framework

As early as 1980's, researchers looked for methods to push computation down into mass storage to process terabytes or even petabytes of data [87]. Following are some of the in-

storage frameworks that have been proposed: DBMS [89,120,202], graph analytics [129,145, 163], HPC applications [197] or general workloads [78,82,106,118,126,140,172,182,187].

The main difference between databases and other big data applications is that the later usually requires running complex programs on large data structures designed for the purpose, while databases are more specific and focus on running structured queries on relational tables. The shared concerns include how to reduce DRAM requirements, reduce network traffic, and exploit the massive internal flash bandwidth.

5.2.2 General database accelerators

Q100 [204, 205] and its newer variant [162] are general query accelerators based on a programmable spatial-array architecture. Both systems assume inputs and outputs are consumed and produced in the main memory. In terms of executing SQL operators in a data-flow style, AQUOMAN is similar to Q100 and its variant but it addresses the main bottlenecks that both architectures ignored:

1. *Scalability to larger dataset:* Q100's speedup over single-thread software dropped 10X-100X on 1GB TPC-H, and it disappeared almost completely in comparison to multi-threaded software [204]. The functional tiles for sort and join in Q100's ASIC prototype can handle up to 1024 records at 315MHz on a 256-bit datapath (10.08GB/s) [204, 205]. This forces Q100 to divide-and-conquer large input tables to a huge number of small partitions which causes poor scalability. We have drastically improved the sorter and the join functional units in AQUOMAN. Our FPGA sorter can stream-sort *1GB data* at 12GB/s, and *256GB data* at 6GB/s if there is enough DRAM accessible to the sorter. We ran AQUOMAN on a 1TB TPC-H and still showed speedup over a 32-thread software baseline.

2. *Routing between functional tiles:* Q100 architecture [205] is built around a complex 2D-mesh network-on-chip (NOC), which takes 30-50% of the area and could be challenging to implement in practice. In their more recent work [162], instead of establishing arbitrary connections between heterogeneous tiles, they chose a fixed grid of homogeneous core. The routing simpler but now each tile needed the capabilities of

all the heterogeneous cores of [205]. If the tiles are designed to process big workloads, its size will become too big to be realistic. AQUOMAN addresses this issue using a hybrid solution, which supports the common dataflow with a fixed pipeline of three different programmable units: selection, map, SQL swissknife(join/sort/aggregate).

3. *Missing memory subsystem specification for terabyte dataset:* Firstly, Q100 and its variant assume inputs are in-memory, and did not specify the method to access to the underlying mass storage, which is necessary for analyzing terabyte dataset. Secondly, like AQUOMAN, when the entire query cannot be mapped to available functional tiles at once, Q100 and its variant have to fold the query computation into several temporal steps by consuming and producing intermediate results from memory. In such cases, Q100 and its variant evaluated the maximum dataset of 1GB and did not describe the size of memory required for storing intermediate results when evaluating a much larger dataset. AQUOMAN is fully intergrated inside flash drive and assumes inputs are in-storage. To reduce the memory footprint, AQUOMAN only stores the join keys and the row index of tables temporally in DRAM, not materializing entire intermediate tables. AQUOMAN also tries to keep row index on flash whenever possible. For 1TB of TPC-H, without optimizing query planner for AQUOMAN, a maximum of 40GB of DRAM is needed for evaluating a large join. With 16GB of memory, we can run 15 out of 22 TPC-H queries on AQUOMAN. Therefore, for large real-life databases, Q100-like design will either be bottlenecked by external disk bandwidth or have to bear the significant power consumption of DRAM-based systems.

Besides, our system advances designs of some critical operator tiles, especially the high-throughput hardware sorter, while Q100 exposes no such microarchitecture internals.

Other work notice the lack of instruction level parallelism in database workloads. Hence, they replace standard x86 cores with more smaller power-efficient hardware threads: Oracle's RAPID [45, 51] has a rack-scale many-core system specialized for big data analytics. At its core sits a power-efficient general-purpose processor aided by hardware acceleration for data movement and data partitioning. Unlike AQUOMAN's streaming model, the execution model of RAPID is essentially running map-reduce on many cores. Only very primitive

SQL operators, bit-vector load and filter, are hardware-accelerated and exposed as special CPU instructions. Mondrian Database Engine [91] employs a similar approach but uses general-purpose cores with SIMD extension as a near-memory processor (NMP) on the logic-layer of a stacked Hybrid-Memory Cube (HMC). While fitting into a restrained power budget, no hardware accelerators for SQL operators are used by Mondrian.

5.2.3 Accelerators for certain database operators

Examples of research focused on implementing specific database operators in hardware include: selection [65, 120, 203, 206], hash join [108, 139], sort-merge join [65, 72], group-by aggregation [203], pattern matching [175, 190], and table histogram generation [119]. Most of these accelerators are attached to memory while a few operate in storage [119, 120, 203]. Operator-specific accelerators assist host-side query execution by task offloading. Our work may use similar operator implementation but our focus is on entire query execution in storage.

FCAccel [202] aggregates SQL accelerators for selection, data aggregation, and hashing on a PCIe-attached FPGA. Like AQUOMAN, FCAccel allows stream processing of selection and aggregation, but used a different technique by dividing tables into small data segments buffered in DRAM. FCAccel is reported to have similar performance as MonetDB running on RAM-disk. FCAccel proposes a collaborative solution with the DBMS software for two-way join. Using a custom query FCAccel shows two tables can be filtered and pre-aggregated and later hash-joined by the host opportunistically. Unlike AQUOMAN, it does not offer a plan to offload multi-way joins.

5.2.4 Query-specific reconfigurable accelerators

To avoid the complexity of designing a general query accelerator, some researchers propose to reconfigure FPGAs for a specific query. SQL operators are implemented as hardware libraries in advance, and are then called and assembled for a particular analytical workload. For example, [79, 201, 218] provide flexible hardware templates for common database operators, while [162] proposes a CGRA architecture where reconfigurable tiles are organized

in a systolic manner. The cost of this methodology comes from both the reconfiguration overhead as well as the requirement of using reconfigurable hardware. Baidu [173] has a hybrid solution; certain fixed-function tiles are connected by default in a way that is similar to Q100, but some tiles are reconfigured on demand.

Chapter 6

Dataflow map of a query

We will first discuss the anatomy of query processing on tables and then describe how these steps are mapped on AQUOMAN.

6.1 Single table query

First, consider the query over a single table `sales_transactions` shown in Figure 6-1.

```
SELECT
    department,
    sum(price*(1-discount)) as netsale,
    sum(price*(1-discount) * (1+tax)) as revenue,
FROM sales_transactions
WHERE saledate<=’2018-12-01’
GROUP BY department;
```

Figure 6-1: Aggregate Query

The columns of the `sales_transactions` table are <`transactionID, department, saledate, price, discount, tax`>. Each row corresponds to a purchase identified by a unique `transactionID`, which is the primary key for this table and the cheapest way to refer to its rows. From a semantics perspective, the query of Figure 6-2 should return the net sale and revenue of each department before 2018-12-01. To produce such an answer, the DBMS typically makes what is called a query plan, for example:

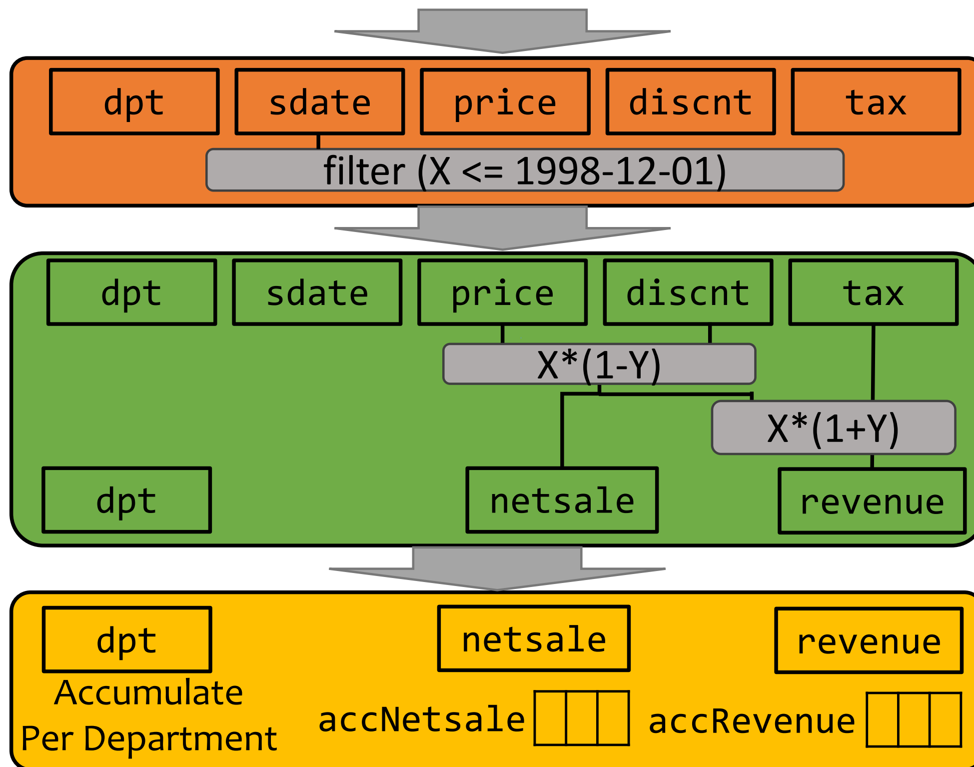


Figure 6-2: Dataflow of an Aggregate Query

1. Filter all rows of the table verifying a predicate: here the `saledate` value should be smaller than 2018-12-01.

2. Produce a new intermediate table of three columns <`department, netsale, revenue`>. Each row of this intermediate table is computed purely from each row selected in Step 1. The department value is directly reported from the incoming row, while the net sale and revenue values are simple arithmetic computation based on the price, the tax and the discount value of the input row.

3. Produce the output table by aggregating the data in the intermediate table, grouped by department.

Note that the first two operations are *map*: they apply functions on each row independently. The last step aggregates data coming from different rows. Those 3 steps can be thought of as a dataflow graph which define how rows of the input table contribute to the query's answer (See Figure 6-2). Actually, the dataflow of this particular query illustrates a common plan of row filtering, intermediate table generation and final reductions. Of course

the functions used for row filtering, table creation and reductions are query specific.The commonality and high value of this fixed but parameterized dataflow in analytics query processing makes it possible to design a fixed accelerator to process such queries efficiently.

6.2 Join – a multiple table query

Suppose the `sales_transactions` table has an extra column to indicate the purchased item. The purchased item is represented as the `inventoryID`, which is the primary key of another table, the inventory table. The `inventory` table has many columns: `<inventoryID, category, quantity, productname, ...>`, where category represents the type of an item.

The following query (Figure 6-3) computes the total sale of `shoes` sold after "2018-03-15".

```
SELECT sum(price) as shoe_sales
FROM inventory as ti, sales_transactions as ts
WHERE ti.invtID=ts.invtID
    and ti.category="Shoes" and ts.saledate>'2018-3-15';
```

Figure 6-3: A Join Query

This query needs to compute a so-called inner equi-*join* on the tables `inventory` and `sales_transactions`. Typically, this join query would be processed as follows:

1. Select all the rows that have category "shoes" in the `inventory`.
2. Produce an intermediate table `<transactionID, inventoryID>` from the `sales_transactions` table; to get all the items (referred to by `inventoryID`) that were sold after 2018-03-15.
3. Merge the two intermediate tables produced by the two previous step: every items that are shoes (intermediate table out of Step1) is filtered based on if there exists a sale of that item within the date specified (intermediate table from Step 2).

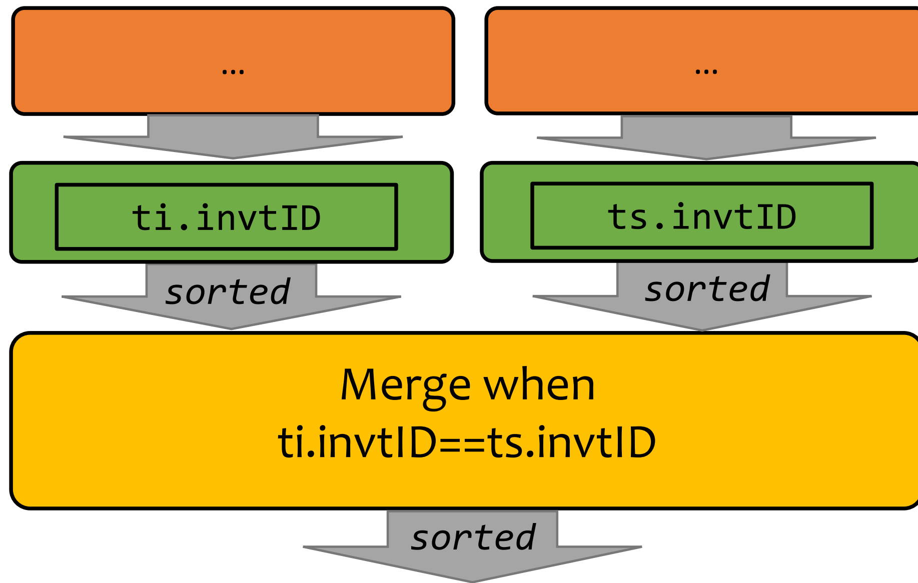


Figure 6-4: Dataflow of a join query

Notice that the Step 1 and 2 are similar to the steps in the previous example; however in this example they are working on two different tables. In contrast, Step 3 does not seem to fit into the fixed dataflow illustrated in the previous example: it merges data from two intermediate tables.

If we assume that the intermediate tables are generated sequentially and stored in the accelerator DRAM, then a streaming sorter can be placed between the producer and the DRAM, to make the joining easy. Typically these intermediate data (the keys involved in the join) are small enough to be stored in few Gigabytes for Terabytes datasets. In rare cases where the intermediate tables is bigger than the accelerator DRAM, AQUOMAN would relinquish processing to the host. Joins require fast *hardware sorters* to keep up with the streaming rate of the underlying storage. Two-way join generalizes to multi-way join by iteratively storing the sorted intermediate tables in the accelerator DRAM.

Chapter 7

Overview of AQUOMAN Architecture

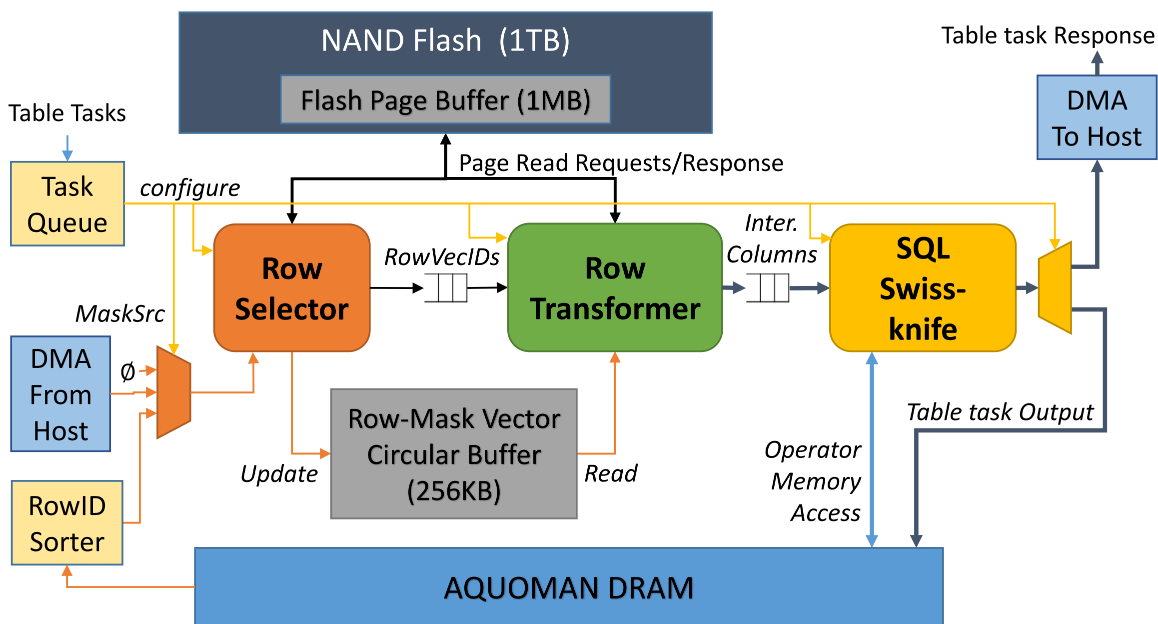


Figure 7-1: Overall Architecture of AQUOMAN

AQUOMAN targets accelerating column-oriented databases like MonetDB. We chose MonetDB because in our TPC-H benchmark evaluation, on average MonetDB was 2X faster than a commercial row-oriented database. In column-oriented database systems, a relational table is stored as a collection of column files. Each column file stores a sequence of column values in ascending row order in either compressed or uncompressed format.

A modern flash drive has huge I/O bandwidth which can easily produce more than one column value per "data beat". For example, a flash hardware controller running at 125MHz

with 4GB/s bandwidth is able to produce 32 bytes - equivalent to 8 32-bit column values - per clock cycle.

To allow line-rate data processing, AQUOMAN processes the column data files as a collection of *Row Vector*s, which consists of 32 column values of consecutive rows, indexed by *Row-Vector ID*. A bit-vector that marks which rows have been selected for processing is also stored as part of the table. The overall architecture of AQUOMAN is shown in Figure 7-1.

The heart of AQUOMAN consists of 3 accelerators *Row Selector*, *Row Transformer* and a *SQL Swissknife*, corresponding to accelerators for the three kind of dataflow operators identified in the previous section. AQUOMAN also relies on one extra *Sorter* to keep the intermediate streams ordered on the required keys.

The *Row Selector* generates the bitvector masks used to efficiently select the input table data (see Section 8 for more details on the expressivity of the *Row Selector*). The columns of the rows that have not been masked and are necessary to compute the intermediate table are then streamed to the *Row Transformer*. The *Row Transformer* is composed of a collection of Processing Elements organized to apply a stateless function on each row to produce a new intermediate table. Finally, the generated rows are fed into the *SQL Swissknife*. The *SQL Swissknife* contains accelerators to perform the standard SQL operators: accumulate, sort, merge, computes the biggest k values . . .

The *SQL Swissknife* is equipped with a direct access to AQUOMAN's DRAM, it can leave an intermediate reduced table in it, or consume an intermediate table from it. We will see the usefulness of that patterns when discussing the acceleration of joins. The dataflow between the three accelerators of AQUOMAN is fixed - the generality and programmability of AQUOMAN comes from the predicates the *Row Selector* applies, the functions the *Row Transformer* computes, and the operators the *SQL Swissknife* runs.

Architecturally speaking, the *Row Transformer* directly streams the intermediate table to the *SQL Swissknife* without materializing it in DRAM. In the benchmark we evaluated, this drastically reduced the need of DRAM for AQUOMAN.

7.1 Multi-SSD AQUOMAN

When input data are stored on many SSDs because of the need for higher capacity, performance or fault tolerance, the database software can process the query without changing its processing model because it can simply read the input data from the centralized page cache. AQUOMAN can handle multi-SSD scenario with a similar approach by devising a small DRAM buffer where input data from different SSDs are gathered.

For a *Table Task* involving a single input table that would be resident on several AQUOMAN-augmented SSD, we would map the same *Table Task* to all the AQUOMANs and return the partial results to the host, responsible to reduce the computation. In case of a *Table Task* involving multiple tables, we would need to broadcast the intermediate table outputs across the different AQUOMAN-augmented SSD, either using a PCIE multicast, or using a custom link directly between the AQUOMANs.

7.2 Programming AQUOMAN

7.2.1 Software Interface

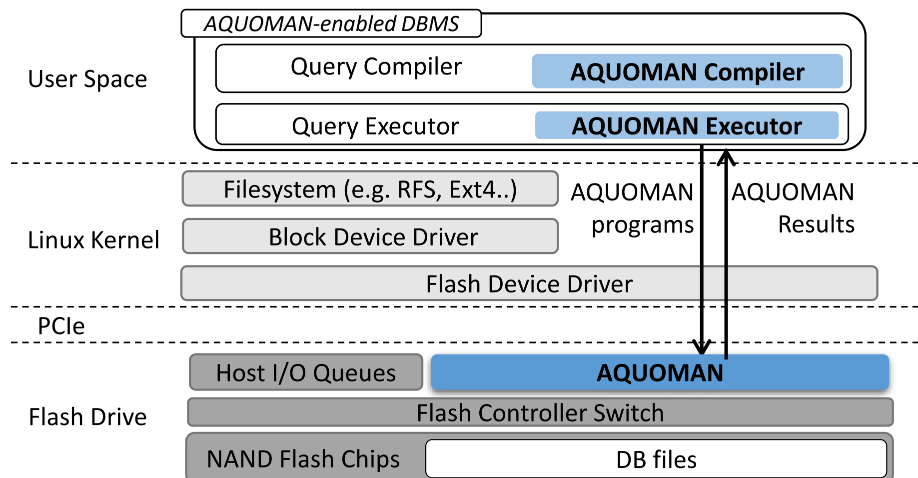


Figure 7-2: System Stack with AQUOMAN

As shown in Figure 7-2, AQUOMAN is located inside the flash drive, so has direct data access to the NAND flash arrays. AQUOMAN and the x86-host can both access NAND flash

simultaneously via a *flash controller switch* inside the flash device, which fairly arbitrates flash commands of *page_read, page_write, block_erase*. User-level applications can access the flash drive via legacy operating I/O stack, such as filesytem and block device drive.

In addition to legacy I/O path, AQUOMAN-enabled software can also send AQUOMAN programs to AQUOMAN inside the flash drive, which directly reads the required database files, executes the program and returns their result to the host.

In general a SQL query is compiled to a graph of *Table Task*(s). We first describe the structure of a *Table Task*:

```
typedef struct {
   string                     table;
   ProcessingMaskSrc          maskSrc;
   RowSelectionProgram        rowSel;
   RowTransformationProgram   rowTransf;
   SQLOperator                operator;
   OutputDestination          outDest;
} TableTask;
```

Figure 7-3: Table Task Structure

A *Table Task* is described as follows:

- `table` specifies the input table of the *Table Task*.
- `maskSrc` specifies the source of the row processing masks, which is generated by a *Row Selection Program*. It can come from the *Host software*, or from AQUOMAN DRAM if produced by a previous *Table Task*.
- `rowSel` specifies a *Row Selection Program*. This selection mechanism can only compute single column predicates, but it provides a fast layer of selectivity to avoid having to stream all the data to later stages.
- `rowTransf` specifies a straightline *Row Transformation Program*, which is mapped over all the rows to transform each one into a row of the new intermediate table. The columns of the intermediate table may be different from those of the source table.

- `operator` specifies a reduction function in the *SQL Swissknife* as an SQL operation on the output table of *Row Transformation Program*. There are seven operators presented in table 7.1

- `Output` specifies the output destination of the *Table Task*, which can be either AQUOMAN or the Host.

Operator	Description
`NOP`	Pass through the output table of *Row Transformation Program*
`AGGREGATE`	Summarizes columns of a table with `min, max, sum, cnt`
`AGGREGATE_GROUBBY`	Summarizes columns of a table with `min, max, sum, cnt` per group
`SORT`	Sort a single-column table of key-value pairs by key
`MERGE`	Merge two single-columnn tables of sorted key-value pairs
`SORT_MERGE`	Sort a single-column table of key-value pairs and merge with another single-column table of sorted key-value pairs
`TOPK`	Returns the biggest k values of a single-column table

Table 7.1: SQL (Sub)Operators

For simple queries such as the *Aggregate Group-By* query of Fig 6-1, it should be clear from the previous section that only one table task is needed.

For more complex queries, AQUOMAN programs can have multiple *Table Task*s, each of them run sequentially using an SQL operator in *SQL Swissknife* that consumes the data left by the previous *Table Task* in the AQUOMAN's DRAM (See Sec. 8.4).

For example, to accelerate the join query (Figure 6-4), the user can create the three *Table Task*s (Figure 7-4) and the associated dataflow graph as shown in Figure 7-5.

Since executing a single *Table Task* on AQUOMAN can saturate the flash bandwidth, executing *Table Task*s sequentially is sufficient to keep up the line rate. AQUOMAN records the intermediate results for the join in its DRAM.

7.2.2 Query-planning for AQUOMAN

For a good integration of AQUOMAN in a DBMS, we will need to modify the query-planner to identify maximal sub-trees of the query-plan that have shapes that can be mapped to

```
auto tabletask_0 = TableTask{
  .table      = "inventory",
  .maskSrc    = RowSelectionProgram,
  .rowSel     = [predicate: category == "shoes"],
  .rowTransf  = [in: inventoryID][out: inventoryID],
  .operator   = NOP,
  .output     = AQUOMAN_MEM_0};
auto tabletask_1 = TableTask{
  .table      = "sales_transactions",
  .maskSrc    = RowSelectionProgram,
  .rowSel     = [predicate: saledate> 2018-03-15'],
  .rowTransf  = [in: inventoryID][out: inventoryID];
  .operator   = SORT_MERGE[with AQUOMAN_MEM_0],
  .output     = AQUOMAN_MEM_1};
auto tabletask_2 = TableTask{
  .table      = "lineitem",
  .maskSrc    = AQUOMAN_MEM_1,
  .rowSel     = [NOP]
  .rowTransf  = [in: price][out: price];
  .operator   = AGGREGATE
  .output     = Host};
```

Figure 7-4: JOIN query *Table Tasks*

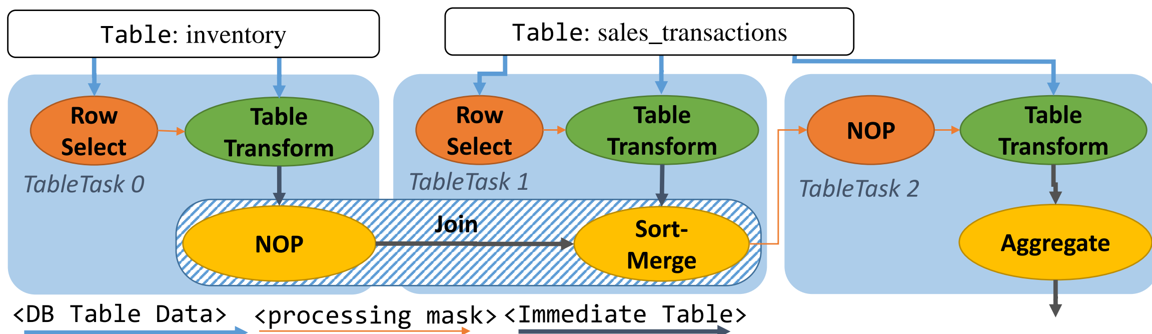


Figure 7-5: JOIN query data-flow graph

*Table Task*s, and run those parts on AQUOMAN. A query-optimizer is also needed for AQUOMAN to optimize some tradeoffs in any such mapping; commuting of filters, ordering of multi-way joins, copying vs. sequentializing the use of an intermediate table. We consider such full-blown compiler work premature before establishing the efficacy of AQUOMAN architecture.

7.3 Query-planning for AQUOMAN

A query-plan, a graph of SQL operators has to be mapped into AQUOMAN's fix dataflow stages *Select/Transform/Reduce*. Since any of these stages can be bypassed (treated as an identity operator), we can simply create a *Table Task* for each supported SQL operator. The task would store intermediate tables in AQUOMAN's DRAM if necessary. Special SQL operators that are not implemented on AQUOMAN, would simply be run on the host. Such a naive strategy would probably turn AQUOMAN into a decelerator!

For a good integration of AQUOMAN in a DBMS, we will need to modify the query-planner to identify maximal sub-trees of the query-plan that have shapes that can be mapped to *Table Task*s, and run those parts on AQUOMAN. A query-optimizer is also needed for AQUOMAN to optimize some tradeoffs in any such mapping; commuting of filters, ordering of multi-way joins, copying vs. sequentializing the use of an intermediate table. Such tradeoffs depend on the amount of DRAM available and the bandwidth of the mass storage, etc. We consider such full-blown compiler work premature before establishing the efficacy of AQUOMAN architecture. Still we have modified MonetDB's query planner to identify *Table Task*s from unmodified query plans, and integrated it with a trace-based AQUOMAN simulator. A full *Table Task* compiler with a plan optimizer is not done; however a number of TPC-H queries are hand-coded to *Table Task*s and evaluated on FPGA.

Chapter 8

AQUOMAN Microarchitecture

To execute a *Table Task*, AQUOMAN first configures the *Row Selector*, the *Row Transformer* and the *SQL Swissknife* using the parameters of the first *Table Task* in the task queue. Before processing a *Row-Vector ID*, the *Row Selector* reserves a Row-Mask Vector slot in the Row-Mask Vector Array in circular order. It notifies the *Row Transformer* by sending it the *Row-Vector ID*.

The *Row Transformer* collects the *Row Vector*s of the base table, and applies a table transformation on it to produce the *Row Vector*s of the intermediate table. The *Row Transformer* then releases the slot in the Row-Mask buffer and passes the *Row Vector*s of the intermediate table to the *SQL Swissknife* to apply the specified SQL operation on the intermediate table. The output is written into AQUOMAN DRAM.

The AQUOMAN runs the three accelerators simultaneously in a pipeline fashion, as long as it can reserve a slot in the row-mask vector array. The maximum number of in-flight *Row-Vector ID*s is determined by the depth of the flash command queue, which determines the size of the Row-Mask Vector Circular Buffer. For example, for a flash controller with a command queue of depth of 128, the Row-Mask Vector Circular Buffer needs to hold a maximum of 128×8K rows of 1-byte elements or 32K 32-element *Row Vector*s.

8.1 Row Selector

The *Row Selector* is a *vector* unit in charge of evaluating the predicate for selection. It accepts predicates in the form:

$$Pr = F(CP_0, \ldots, CP_{n-1})$$

where CP_i is a comparison or an equality to a constant for the value in column i, and F is a simple boolean function. For example $(price > 25)\&(data < 2019-11-26)$ is representable with $F = \&$ and $CP_0 = price > 25$ and $CP_1 < 2019-11-26$. The maximum number of permissible CP terms in a filter predicate is determined by the number of *Column Predicate Evaluators*; 4 to 6 evaluators are enough for most of the filter predicates in TPC-H. When the *Row Selector* cannot compute a predicate, e.g. predicates which require more than one column, or regular-expression filtering, it forwards them to *Row Transformer*, the next stage in data-flow.

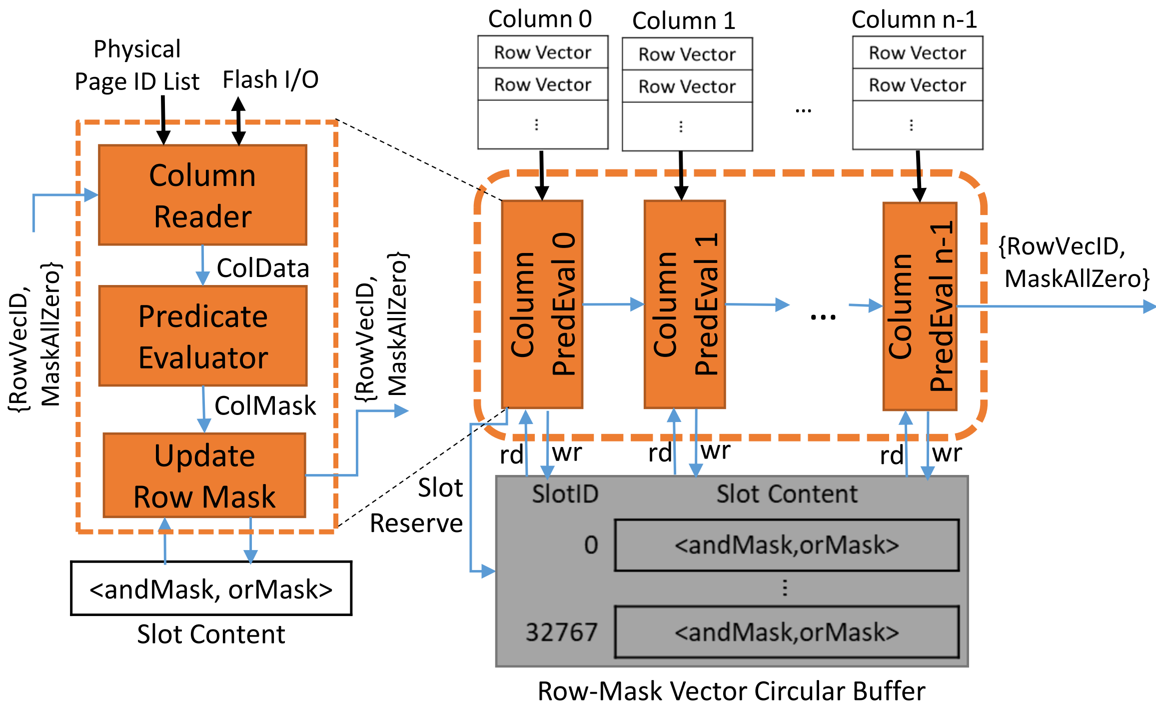


Figure 8-1: Architecture of Row Vector Selection

To have the *Row Selector* compute such a predicate, we first put F in a minimal sum-of-product (SOP) form. Then the *Row Selector* computes as follow: it starts with a constant 0

in an *orbv* accumulator and a constant 1^* in an *andbv* accumulator. [1]. It accumulates the first product of the SOP predicate, one term at a time in the *andbv* accumulator. When it reaches the end of the first product, it accumulates the complete first product *andbv* into the partial sum: $orbv = orbv|andbv$ and reset the product accumulator: $andbv = 1^*$. This way it accumulates all the products one by one in the partial sum.

The *Filter* predicate in SOP form only requires two *one-bit of memory storage* per row for accumulating intermediate predicate results, one for *AND operation* and one for *OR operation* making it cheap to implement.

We envision that time-sharing one *Column Predicate Evaluator* between several predicates could be useful for more complex predicates, but this has not been done yet.

Finally, the computation based on the SOP form - using only minimal memory footprint - may be sometimes suboptimal in the sense that it may require reevaluating the same column predicate twice (e.g $\underline{CP_0} \cdot CP_3 + \underline{CP_0} \cdot CP_2 \cdot CP_6$).

8.2 Row Transformer

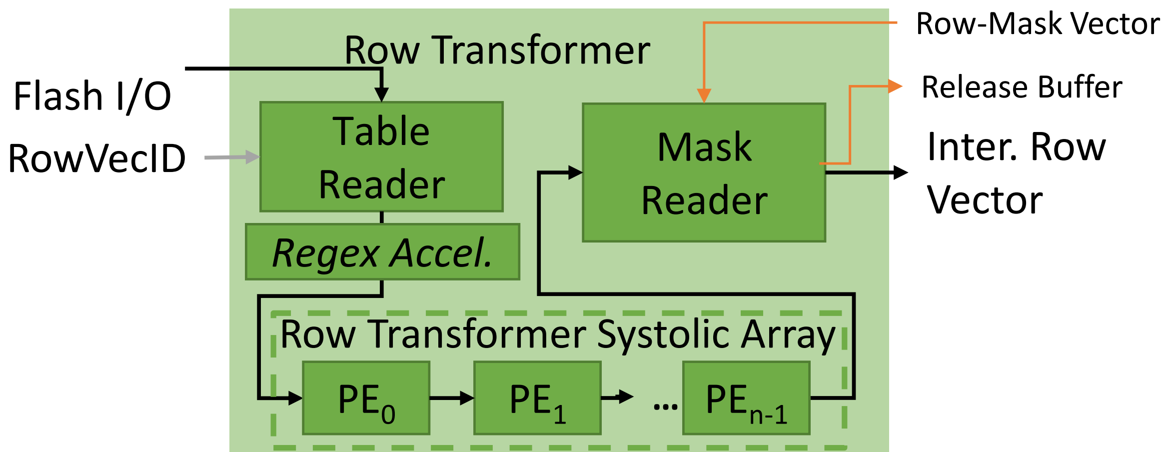


Figure 8-2: Row Transformer Architecture

The *Row Transformer* has three components: the *Table Reader*, the *Row Transformation Systolic Array* and the *Mask Reader*, as shown in Figure 8-2.

[1]Note that it is a vector processor, hence the suffix "bv" for bitvector, but one can read the section with in mind a *Row Selector* of vector-width 1 without loss of generality

```
SELECT l_quantity as qty,
    l_extendedprice as base_price,
    l_extendedprice*(1-l_discount) as disc_price,
    l_extendedprice*(1-l_discount)*(1+l_tax) as charge,
FROM  lineitem WHERE l_shipdate <= date '1998-09-01';
```

Figure 8-3: SQL Query Example for Table Transformation

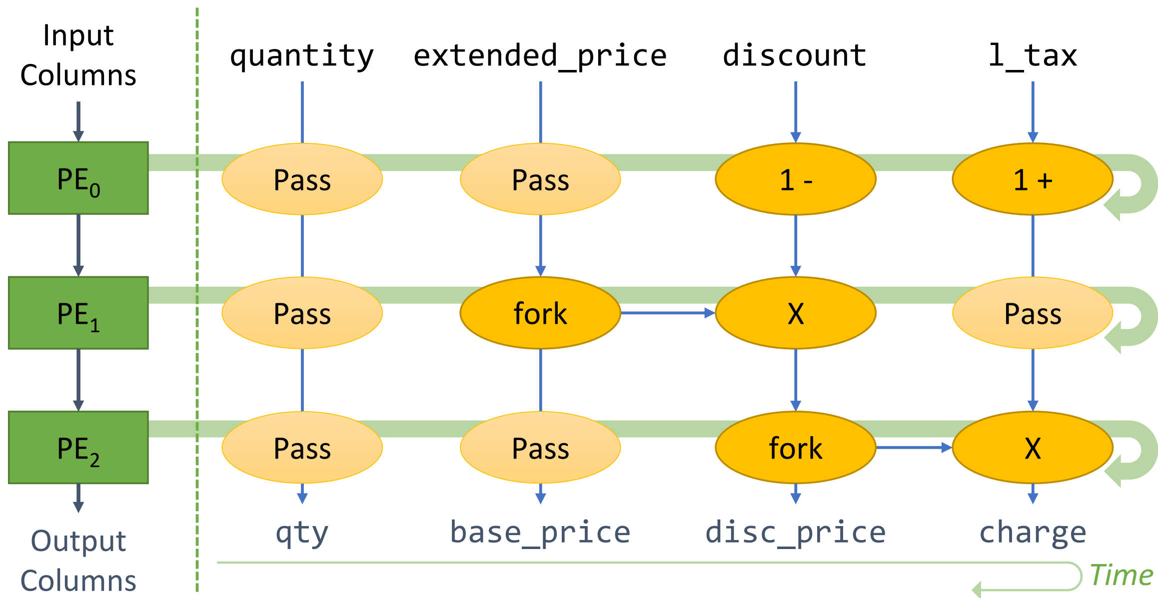


Figure 8-4: Data-Flow Execution Diagram of Table Transform

The Table Reader initiates reading the flash drive when it receives a *Row-Vector ID* from the *Row Selector*. It skips reading a flash page if all its *Row-Vector ID*s are marked as zero in the bitvector mask. The Table Reader streams out *Row Vector*s to the *Row Transformation Systolic Array* in increasing order of *Row-Vector ID*s; within each *Row-Vector ID*, streaming is done from the leftmost column to the rightmost one.

Inside the Table Reader there is also a *Regular-Expression Accelerator*. It pre-processes variable-sized (string) columns to a one-bit column (true/false). The accelerator has a 1MB memory to store the strings of the column. 1MB is sufficient to cover many cases where the strings have a small domain, for example, the "country name" column.

The *Row Transformation Systolic Array* applies a *mapping* function to each row of the input table to produce an intermediate output table. That is, only column data of the same row are taken together to calculate columns of a new row.

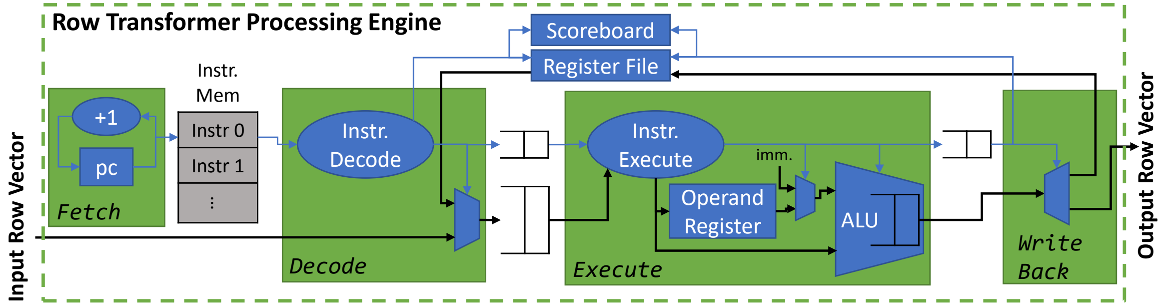


Figure 8-5: Micro-architecture of a Row Transformer PE

The *Row Transformation Systolic Array* is a systolic architecture where the transformation function implied by a query is mapped to an array of PEs. Since the Table Reader streams out *Row Vectors* per *Row-Vector ID* in a fixed order, we can draw a data-flow graph of transformation steps from the input columns to output columns. For example, the mapping of the data-flow graph for the query in Figure 8-3 is shown in Figure 8-4. An AQUOMAN compiler can balance the transformation data-flow graph by inserting PASS nodes (NOPs), so that it can be mapped onto the PE. It can share common subexpressions used in computing several output columns by inserting FORK nodes (`Copy` Instruction). The compiler must maintain the invariant that the nodes of the compiled data-flow graph can only have data transfers to their south and/or east neighbor(s). In particular, no cycles are allowed in the dataflow graph.

Each PE performs transformation steps for multiple output columns in a circular schedule. It also produces new *Row-Mask Vectors* for filtering (sub)predicates which have not been processed by the *Row Selector*. The Mask Reader then merges the old *Row-Mask Vector* (produced by the *Mask Reader*) and the new *Row-Mask Vector* and passes it to *SQL Swissknife*. Finally, it releases the slot in the Row-Mask Vector Circular Buffer.

Each processing engine (PE) in the *Row Transformation Systolic Array* is a simple 4-stage integer arithmetic vector processor with no branch instructions or data memory (Figure 8-5). It implements a simple 32-bit instruction set described in Table 8.1. Each PE has 7 general purpose registers (`rf[1]`,..., `rf[7]`), an operand fifo (`opReg`). Finally it has a special fifo, which can be accessed as a register (`rf[0]`), hardwired to be read as input fifo and written into as the output of the PE.

The instruction memories of the PEs are initialized by the *Table Task*. Since there are no

<table>
<tr><th>Opcode</th><th>AluOp</th><th>Descr.</th></tr>
<tr><td>Pass</td><td></td><td>rf[rd]<=rf[rs]</td></tr>
<tr><td>Copy</td><td></td><td>rf[rd]<=rf[rs]; opReg<=rf[rs]</td></tr>
<tr><td>Store</td><td></td><td>opReg<=rf[rs]</td></tr>
<tr><td rowspan="7">ALU(Imm)</td><td>Add</td><td>rf[rd]<=rf[rs] + <OpReg|imm>.</td></tr>
<tr><td>Sub</td><td>rf[rd]<=rf[rs] - <OpReg|imm></td></tr>
<tr><td>Mul</td><td>rf[rd]<=rf[rs] * <OpReg|imm></td></tr>
<tr><td>Div</td><td>rf[rd]<=rf[rs] / <OpReg|imm></td></tr>
<tr><td>EQ</td><td>rf[rd]<=rf[rs] == <OpReg|imm></td></tr>
<tr><td>LT</td><td>rf[rd]<=rf[rs] < <OpReg|imm></td></tr>
<tr><td>GT</td><td>rf[rd]<=rf[rs] > <OpReg|imm></td></tr>
</table>

Table 8.1: PE Instruction Set

branches, the program counter (PC) will always increment by 1 and roll back to 0 at the end of the program. The size of the instruction memory of each PE should be bigger than the number of nodes in the transformation diagram, which equals the number of input columns to be transformed.

Once an instruction is fetched and decoded, the input *Row Vector* is read either internally from the Register File or externally (`rs==0`). The *Row Vector* is placed either in an operand register waiting for the second operand, or sent to a pipelined ALU with its other waiting operand. The Execute stage performs the operation and in the write-back stage, the output of the ALU is either written in the register file or streamed out (`rd==0`). The Register File is used only for data passing vertically between nodes when multiple nodes of a vertical slice of the graph are mapped to a single PE. Such a case happens only when the number of PEs exceeds the number of horizontal layers of the data-flow graph.

8.3 SQL Swissknife

The *SQL Swissknife* is configured by the *Table Task*, which takes the intermediate table output from the *Row Transformer*, and applies one of the SQL (sub)operations listed in Section 7.2. Inside the SQL Swissknife is an array of accelerators, whose connection to the external input is configured by the *Table Task* (Figure 8-6). When *Row Vector*s are streamed in, they are tagged with a Column ID which is needed for processing a table of more than

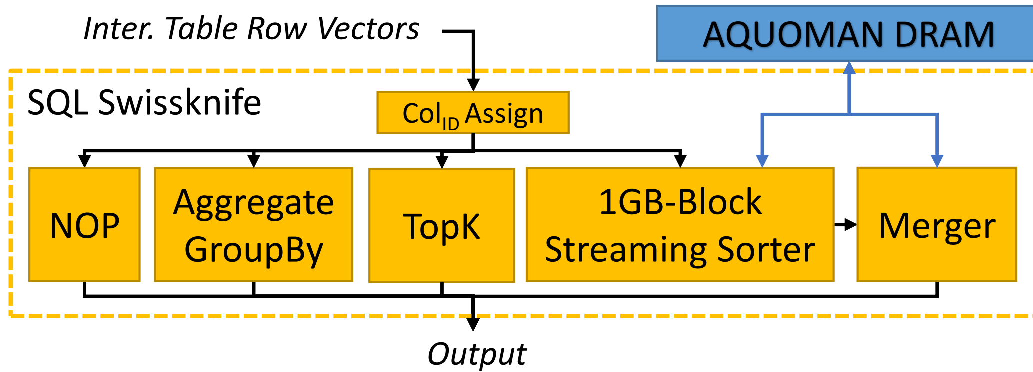


Figure 8-6: SQL Swissknife Architecture

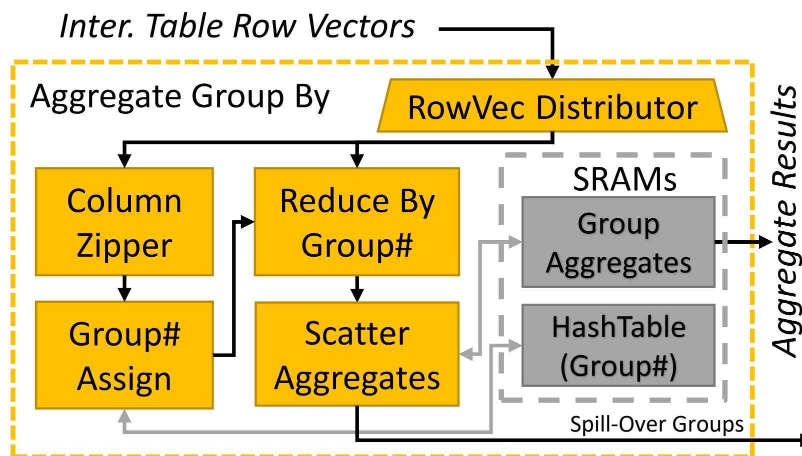


Figure 8-7: Aggregate-GroupBy Accel.

one columns (e.g the input table of an Aggregate GroupBy). Each SQL operation is mapped to its corresponding accelerator(s). SQL sub-operators of `SORT`, `MERGE` and `SORT_MERGE` are mapped to two serially linked accelerators: the Streaming Sorter and the Merger. For `SORT` and `MERGE`, one of them is configured as a NOP.

New SQL operation accelerator can be added into SQL Swissknife with or without DRAM access as needed. In our current version of the *SQL Swissknife*, only the Streaming Sorter and Merger are connected to the DRAM.

8.3.1 Aggregate GroupBy

The Aggregate GroupBy accelerator handles grouping rows of the same group identifier into summaries of aggregation attributes of `sum`, `min`, `max`, and `cnt`. It does local *Aggregate*

Group-By operation per *Row-Vector ID*, and then scatters and updates the local group aggregates to the corresponding global aggregates stored in SRAM.

As shown in Figure 8-7, the Aggregate GroupBy accelerator separates *Row Vector*s of columns into two different streams using their Column IDs. If *Row Vector*s are used for identifying groups, they are sent to the *Column Zipper*, otherwise they are sent to the *Reduce-By-Group-Number* block, waiting for their groupIDs to be assigned.

Column Zipper zips multiple *Row Vector*s of the same *Row-Vector ID* into a super *Row Vector* named the *Group Identifier Vector*. The *Group Number Assign* component assigns it a Group Number using a hash-table of 1024 buckets. Each bucket can hold at most one group identifier of maximum size of 16B. New group numbers are assigned in an increasing order from 0 to 1023. In case of a hash collision of two group identifiers, one group is kept and the other one is marked as a spill-over group, which is sent to x86 host for processing. (more on this in Section 6.5)

After a Group Identifier Vector is given a Group Number, it is sent to the *Reduce By Group Number* block, in which its corresponding *Row Vector*(s) are reduced per group. The reduction results of `sum, min, max, cnt` are scattered into an SRAM and accumulated with the global aggregates indexed by group number. Each aggregate slot can store aggregates for 8 individual columns.

Since the SRAMs are expected to scatter/gather a maximum of 32 addresses per request, we have partitioned the SRAM into 32 partitions by striping the address space, allowing bigger bandwidth through banking. If addresses per scatter or gather request are uniform, we can pipeline the requests without many memory stalls.

8.3.2 TopK

The TopK accelerator takes in a stream of *Row Vector* from a table and keeps the biggest k rows of the stream. In software, the *TopK* operation is computed using minHeap, which cannot be easily pipelined in hardware. Instead, we use a chain of *Vector Compare-And-Swap* blocks (VCAS) to store the k biggest elements, as illustrated in Figure 8-8. Each VCAS stores n elements where n is the input vector size. When a vector of size n is fed

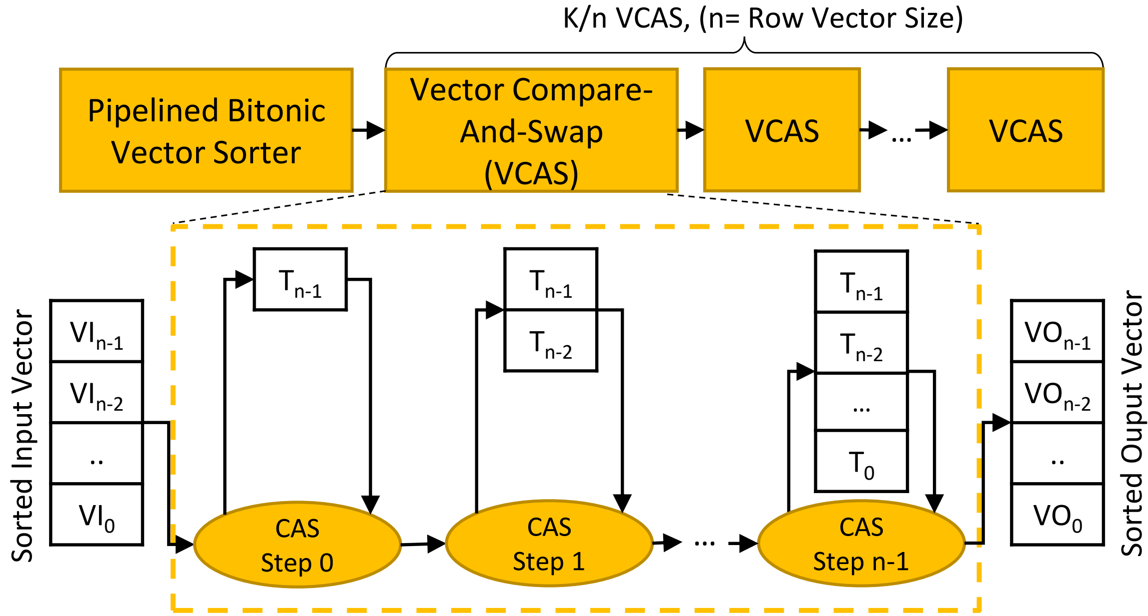


Figure 8-8: TopK Accelerator

into a VCAS, VCAS compare-and-swaps it with the n elements stored inside the VCAS, where the bigger half of $2n$ elements are kept, and the smaller half is streamed out. We can daisy-chain k/n VCAS to keep the top k elements.

Before sending it into the chain of VCAS, the input vector is first sorted using a pipelined bitonic sorter. This is done because the pipelining of VCAS operation for sorted vectors can be done more efficiently, as shown in Figure 8-8. Each VCAS operation of two sorted vectors of size n can be divided into n steps of compare-and-swap element-wise, as shown in Algorithm 1. The ith element-wise CAS step generates a partial result of the top-i vector

Algorithm 1 Vector Compare-and-Swap

Variables: $InVec$: Input Vector sorted in ascending order
$TopVec$: Top-n Vector sorted in ascending order

```
    tailIn = tailTop = n − 1
 2: for i in 0..n − 1 do
        if InVec[tailIn] > TopVec[tailTop] then
 4:         swap(InVec[tailIn], TopVec[tailIn])
            tailIn = tailIn − 1
 6:     else
            tailTop = tailTop − 1
 8:     end if
    end for
```

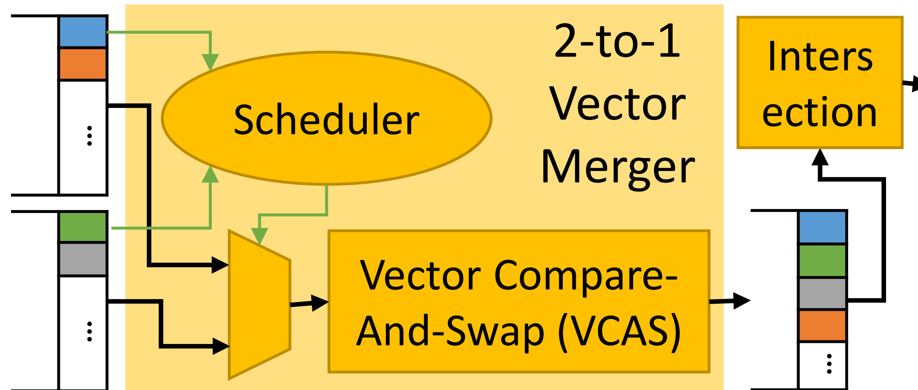


Figure 8-9: Merger Architecture

which can be consumed by the *i*th step of the next input vector immediately. Therefore, a VCAS hardware can be pipelined properly. Each *i*th pipeline stage of the VCAS takes up reasonably small hardware resources with one pair of `i-to-1` muxes for compare, and one pair of `i-to-1` muxes for data update.

8.3.3 Merger

The Merger accelerator outputs the intersection of two sorted list. The Merger accelerator first merges two sorted list into one sorted list using *2-to-1 Merger*, and then passes it through an *Intersection Engine* where the non-intersected part is dropped (Figure 8-9).

In case of duplicate values in the input sources, the merger always tries to alternate the input sources. This way the Intersection Engine only needs a look-ahead of one to decide if it should drop a value or not. Indeed if in the final sorted stream two consecutive values are equal but not coming from the same source, one of them can be dropped knowing that the same value from the other source could not arrive later.

Inside the 2-to-1 Merger, we have the *Vector Compare-And-Swap Engine* which does the merging, and a *Scheduler* which decides which input vectors of the two sorted streams should be fetched. Since items of each input vector are sorted and the input vectors per data stream are sorted, the Scheduler only needs to compare the top items of the two input vectors and send the input vector with the smaller top item to VCAS.

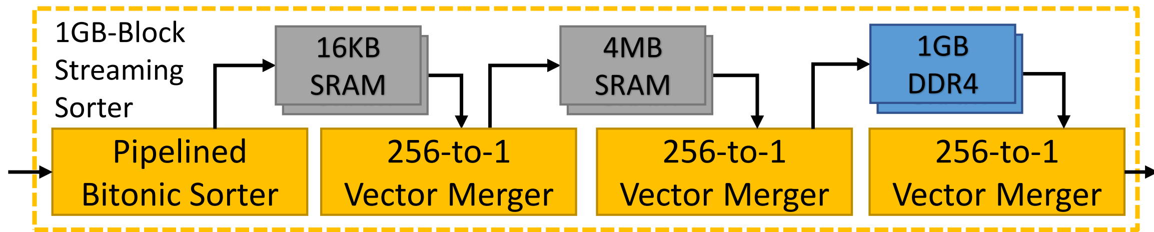


Figure 8-10: Streaming Sorter Architecture

8.3.4 1GB-Block Streaming Sorter

The 1GB-Block Streaming Sorter takes an unsorted stream of input vectors, and outputs a stream of sorted 1GB blocks. The Streaming Sorter consists of a *Pipelined Bitonic Sorter* which sorts 64-byte input vectors, and merge 2^{24} 64 bytes vectors into a 1GB sorted stream using three layers of 256-to-1 Mergers (Figure 8-10).

The first two layers of the 256-to-1 mergers merge 256 64B-blocks to a 16KB block, and 256 16KB-blocks to 4MB-block respectively. They store the immediate results on SRAMs. The last layer merges 256 4MB-blocks to 1GB block using DRAM. The SRAMs and DRAM need to be duplicated per layer to maintain the line-rate of input stream. If the sorter had enough DRAM, it can sort 256GB by folding the last 256-to-1 merging step at the half of the streaming speed.

Each 256-to-1 Merger is constructed using a binary tree of 2-to-1 mergers which were introduced in Section 8.3.3. Since the average of utilization of 2^i 2-to-1 mergers at the same depth i of the binary tree is only 1, we make 2-to-1 mergers at the same depth share the same VCAS component capable of keeping multiple contexts. In this way, we can decrease the size of N-to-1 merger from $O(2N-1)$ to $(O(logN))$ while still able to keep up with the input rate.

To perform sort-merge join, both join-key columns don't have to be totally sorted. As long as one column is totally sorted, a partially-sorted second column can be merged with it at the cost of re-streaming the first one for every 1GB of data stream. This can cause more than 1GB DRAM reads per 1GB flash reads, but is OK because DRAM is an order-of-magnitude faster than flash. In many cases, AQUOMAN doesn't even need to sort the first column, since primary keys are already stored by MonetDB in its internal representation.

8.4 AQUOMAN Memory Management

Because of the fixed dataflow pipeline of AQUOMAN, a SQL query often needs to be broken into multiple *Table Task*s which are executed on AQUOMAN sequentially. AQUOMAN stores the intermediate tables produced by each *Table Task* in DRAM and merges them using subsequent *Table Task*s. AQUOMAN's memory management system only keeps the row indices of tables and join keys in DRAM to compute *multi-way joins*, which allows us to keep the DRAM footprint small. AQUOMAN memory management does not buffer the results of *Aggregate Group-By* and *TopK* operators, because such operators are typically at the end of an SQL execution plan. When such operators are not the last operator of the query, we cannot off-load the part of the query following the *Aggregate Group-By* or *TopK* operator. Such cases are uncommon and AQUOMAN can often accelerate even partially offloaded queries (see Section 10.3).

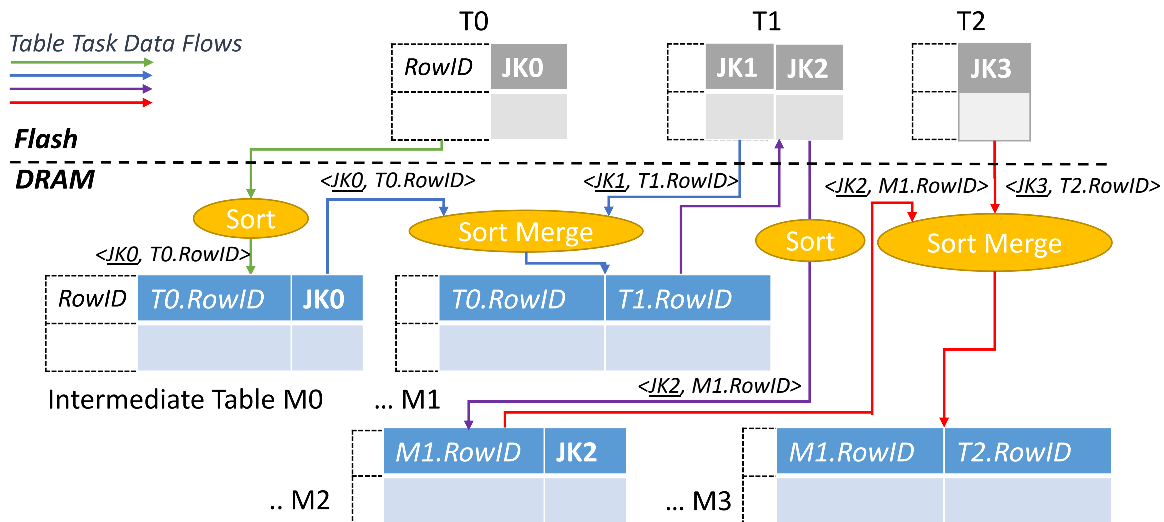


Figure 8-11: AQUOMAN memory operations of a three-way join (*T0.JK0=T1.JK1 and T1.JK2=T2.JK3*)

Figure 8-11 exemplifies how AQUOMAN manage s DRAM for a three-way *Join* query. In the figure there are 3 input tables on the flash drive, T0,T1 and T3, which are to be joined based on *T0.JK0=T1.JK1 and T1.JK2=T2.JK3*. We also have 4 intermediate tables, M0, M1, M2 and M3, which are stored temporarily on DRAM. In our table data structure, each table has its own "virtual" RowID column which are simply the row offsets of table and not materialized on flash or DRAM. The intermediate tables on DRAM can store columns of

join keys and RowIDs which are backward pointers to the rows of other tables.

A RowID column provides index to rows of a table. Such a column is implicit and does not need to be stored in DRAM or flash. A multi-way join is decomposed into two-way joins, where each two-way join is executed using a *sort-merge join* expressed by two *Table Task*s. Each data flow arc which goes into *sort* and *sort-merge* operations carries key-value pairs, where the key field is used for sorting and merging, and the value field has the RowID representing where the join key is read from. The *Table Task*s of each two-way join produces two intermediate tables. The intermediate tables produced by sort *Table Task*s are consumed by their subsequent sort-merge *Table Task*s, and can be garbage collected immediately. The intermediate tables produced by sort-merge *Table Task*s store backward pointers, *i.e.* RowIDs, which are needed for constructing the final result of a multi-way join. And they are stored for the entire lifetime of a multi-way join query.

AQUOMAN also deploys MonetDB-specific optimizations to save memory. MonetDB uses RowIDs to represent the primary keys of tables internally, and for each foreign key column it materializes an additional column of RowIDs referring to the primary keys. MonetDB uses RowIDs to perform join whenever is possible. AQUOMAN is aware of the internal structure used by MonetDB and avoids loading the RowIDs to DRAM whenever possible. Such an optimization opportunity arises when all the primary keys of a table are used for a join operation, i.e. no row of the table has been deleted or filtered out. No join operation is required by AQUOMAN in this case since all foreign keys of the second table are guaranteed to find their matching primary keys. Therefore we can avoid using DRAM and directly construct the join result using the materialized RowIDs on flash.

8.5 Suspending Query Processing on AQUOMAN

There are several reasons why a query may not be completely processed by AQUOMAN:

1. A query has an *Aggregate Group-By* operator in the middle of an execution plan, which breaks references to the base tables on flash.

2. A query does regular-expression filtering on a variable-sized string column which

requires pointer references to a string heap file. When there are many unique strings, such string operations cause random reads to the string heap on the flash and is unsuitable for processing by AQUOMAN.

3. An *Aggregate Group-By* operator in a query generates more groups than what AQUOMAN's SRAM can accommodate.

4. A multi-way join operation in a query produces intermediate tables that exceed AQUOMAN's DRAM capacity.

Conditions 1 and 2 can be detected by examining the query plan, and AQUOMAN can simply suspend processing the query at the appropriate point and pass the intermediate table of results to the host, which can resume processing the query. Since the host will need to access the AQUOMAN SSD when it resumes the query processing, that SSD remains essentially unavailable to AQUOMAN until the query to AQUOMAN has been processed completely. Conditions 3 and 4 can be detected only during query execution. If the database system has an estimate for the size of the intermediate data structure for a specific dataset, it may decide not to offload a part of the query to AQUOMAN. Otherwise, AQUOMAN may use the suspension strategy described next.

For large *Aggregate Group-By* operator, AQUOMAN computes all the hashes but performs the accumulate operation on some buckets in AQUOMAN, while the accumulation for the "spillover" buckets is performed by the host. To not slow down AQUOMAN, the host needs to keep up with the spills generated by AQUOMAN.

For multi-way *Join*s, when the AQUOMAN DRAM becomes full, it keeps sorting 1GB-data-blocks and sends them to the host via DMA. The host completes the join operation by merging these sorted blocks with the sorted data stored in its DRAM.

A natural question to wonder about is how common are these suspensions. As we will show in Chapter 10, 14 out of the 22 queries of TPC-H can be offloaded completely to AQUOMAN with sufficient DRAM. Queries (11,17,18,22) encounter *Aggregate Group-By* operator in the middle and thus, had to be suspended; all except Q22 benefited by partial offloading. There was no benefit to offload queries (9,13,16,20) because they involved regular-expression filtering on a string column.

Seven queries caused spillovers in the *Aggregate Group-By* operation. Only Q18 caused a significant spillover (required ~1.5 billion buckets while AQUOMAN has only 1024 buckets!). Partial offloading of Q18 was still profitable, assuming the host could perform ~200 millions memory lookup-and-accumulates per seconds. With 40GB DRAM in AQUOMAN there were no suspensions due to multi-way *Join*s. A conservative approximation of the effect on performance of memory limitations is discussed in Chapter 10.

8.6 AQUOMAN as a Near-memory Accelerator

AQUOMAN is designed firstly as a near-storage accelerator because we believe that 1) there is a significant number of SQL analytic queries that analyse a vast of amount data and their input data-sets cannot fit in DRAM; 2) When DRAM caching of input data is ineffective, the query speed is bounded by the storage access bandwidth. Therefore the design of AQUOMAN is tailored deliberately for a moderate date stream rate (∼12.8GB/s maximum), which can saturate a modern high-speed NVM drive bandwidth. We also designed AQUOMAN circuit for an FPGA implementation, because FPGA is the common compute fabric for high-performance near-storage accelerator in commercial SSDs [36]. For example, our sort-merger was deeply pipelined using a series of vector compare-and-swap engine (See Section 8.3.2), which may not be needed for an high-speed ASIC design (because ASIC designs are able to close timing for high frequency more easily than FPGAs).

If certain analytic workloads can benefit from caching reused data in DRAM, AQUOMAN design can also be efficiently adopted as a near-memory accelerator. In such a case, we assume AQUOMAN is integrated with a general-purpose processor, which shares the DRAM via the memory controller like an embedded GPU. CPU programs AQUOMAN using *Table Task*s, and provides a input data pointer to memory. And AQUOMAN will execute the query by consuming the input data from DRAM.

A near-memory AQUOMAN should be designed as an ASIC to be able to process at 100-200 GB/s rate, which is the total DRAM bandwidth on a typical general-purpose server. AQUOMAN can be adapted for high-performance ASIC designs by redesigning some of its circuits because they were designed for much slower FPGA clock speed. For example, the

aggressive pipelining of the merger can be relaxed to use less area while still achieving good clock frequency on an ASIC.

If some input data are not DRAM-resident, CPU can also orchestrate data streams for near-meory AQUOMAN by loading input data from secondary storage to DRAM. Essentially CPU acts as a data prefetcher for AQUOMAN, and it should overlap prefetching data with the data processing by AQUOMAN. If a significant portion of input data are read from storage, DRAM is essentially a "streaming cache" for AQUOMAN and the query speed will be bottlenecked by storage I/O bandwidth.

Chapter 9

FPGA Implementation and Evaluation

9.1 FPGA Prototype

We implemented AQUOMAN on BlueDBM [126], where a hardware-accelerated storage device is plugged into the PCIe bus of 12-core Xeon X5670 machine. Each storage device consists of a Xilinx Virtex Ultrascale FPGA development board, VCU108, attached to 1TB of open-channel NAND flash array capable of 2.4GB/s read access and 800MB/s write access. The Xilinx VCU108 FPGA also provides 4GB of DDR4 memory for a maximum bandwidth of 36GB/s.

We synthesized the *Sorter* and the rest of AQUOMAN on two different FPGAs because together their area exceeded the capacity of the VCU108 FPGA. In our AQUOMAN implementation (Table 9.1) the *Row Selector* has 4 Column Predicate Evaluators, and the *Row Transformer* has 4 processing engines each with 8 instructions. Our design meets the timing requirement for 125MHz and provides 4GB/s processing rate for AQUOMAN.

1GB-Block Hardware Sorter: We synthesized the 1GB-block streaming sorter for four data types: 32/64-bit integers, and key-value pairs of 32/64-bit integers. All designs were synthesized with a 512-bit data path and met the timing requirement for 200MHz on Xilinx UltrascalePlus VCU118. Flip-Flops usage was around 40% for each configuration (see Table 9.2).

The area of *Sorter* and AQUOMAN together exceeds the Xilinx VCU118 capacity by 2% but we are confident that with a few area optimization we can fit both of them on a VCU118.

Module Name	LUTs	Flip-Flops	RAMB36	DSP48
Row Selector	42023	36725	0	0
Row Transformer	47859	29660	0	256
SQL Swissknife (w/o sorter)	95077	76823	140	0
FlashPageBuffer	14087	17143	228	0
RowMask	190	41	58	0
VCU108 Total	302398 (56%)	273245 (24%)	448 (26%)	256 (33%)

Table 9.1: AQUOMAN resource usage on VCU108

Element Type	LUTs	RAMB36	URAM
`uint32`	855867 (72%)	1133 (52%)	256 (27%)
↳256-to-1 Merger to 16KB	↳240567 (20%)	↳177 (8%)	↳0 (0%)
↳Vector CAS	↳101476 (8%)	↳0 (0%)	↳0 (0%)
↳Merger Scheduler	↳139091 (12%)	↳177 (8%)	↳0 (0%)
↳256-to-1 Merger to 4MB	↳263610 (22%)	↳291 (13%)	↳256 (27%)
↳256-to-1 Merger to 1GB	↳261400 (22%)	↳505 (23%)	↳0 (0%)
`uint64`	925572 (78%)	1133 (52%)	256 (27%)
`kv<uint32,uint32>`	720183(60%)	1133 (52%)	256 (27%)
`kv<uint64,uint64>`	900087(76%)	855 (40%)	256 (27%)

Table 9.2: Streaming Sorter resource usage on VCU118

Unfortunately VCU118 is incompatible with the custom flash card in BlueDBM.

Although AQUOMAN uses 64-bit key and value pairs as the *Sorter* configuration, we evaluated the streaming sorter for all sorter configurations. As expected, all configurations have the same throughput. Table 9.3 summarizes the performance of the *Sorter* for different input lengths and sortedness using a traffic generator. Hence our *Sorter* meets the goal of keeping up with AQUOMAN processing bandwidth (4GB/s).

Input Length	Input Sortedness		
(GB)	Sorted	Reverse Sorted	Random
1	4.4 GB/s	4.4 GB/s	6.2 GB/s
10	7.9 GB/s	7.9 GB/s	11.0 GB/s
100	8.5 GB/s	8.5 GB/s	11.9 GB/s
1000	8.6 GB/s	8.6 GB/s	12.0 GB/s

Table 9.3: 1GB-Block Streaming Sorter Throughput

9.2 Query Evaluation Setup

9.2.1 Evaluation Data-Set

We used the TPC-H synthetic data-set with a scaling factor of 1000, generating 1TB of tables. We have loaded the data-set on MonetDB-11.27.9, whose column files are the inputs of AQUOMAN.

9.2.2 Baseline Setup

The baseline is MonetDB software (11.27.9) running on single server with a 16 hyper-threaded cores of Intel Xeon E5-2690(2.90GHz) and 128 GB of DRAM. For data storage, the server has a RAID-0 consisting of five 1TB Samsung EVO 970 NVMes, which is throttled to 2.4 GB/s to match the flash bandwidth of BlueDBM storage device. MonetDB does not implement a page buffer bool for caching hot pages, instead it relies on page cache provided by Linux kernel to take advantage of page locality.

When evaluating each query, we use cgroup to limit MonetDB with a range of the maximum DRAM sizes, which is used for both page cache and intermediate tables by query processing. For each query evaluation, we purge the page cache first and then run the query four times. The first run time is recorded as cold run, the average of the last four run times is recorded as hot run.

9.2.3 AQUOMAN Setup

We have hand-coded SQL queries as *Table Task*s. Each *Table Task* execution except Join do not need to use DRAM for buffering the tables between Table Tasks. AQUOMAN has a maximum of 4GB DRAM buffer for *Table Task* outputs.

9.3 Single-Table Query Evaluations

9.3.1 Query A: Filter Operation with High Selectivity

The following query is a variant of Q06 in TPC-H benchmark.

```
SELECT sum(l_extendedprice * l_tax) as total_tax
FROM lineitem
WHERE l_shipdate >= date '1994-01-01'
  and l_shipdate < date '1994-01-01' + interval '1' year
  and l_discount between 0.06 - 0.01 and 0.06 + 0.01
  and l_quantity < 24;
```

This query evaluates a complex filter operation followed by an aggregate on the `lineitem` table of 6 billion rows. The total input data-set is 178GB, and *Row Transform* and *Row Select* do not share input columns. The selectivity of this query is high and only 1.9% of rows are actually selected for aggregation. Due to the high selectivity, the memory footprint of MonetDB for this query is only around 3GB.

Figure 9-1 shows the evaluation result of the query. We can see that MonetDB can process the query across the full range of DRAM capacities due to the small memory footprint. Yet the input data-set of the query is 178GB, which cannot fit in the Linux page cache capacity. This query needs to scan the large input data-set which thrashes the LRU-based Linux page cache. Although on cold runs MonetDB also experiences a similar level of page-cache misses to hot runs, we hypothesize that the cost of finding misses in a fully populated page-cache is much larger because operating system needs to transverse a bigger page-cache index structure to find nothing.

For this query AQUOMAN does not need any DRAM since we are only processing a single *Table Task*. The result shows that AQUOMAN is bottle-necked by BlueDBM's storage drive which streams data at 2.4GB/s. Compared to the baseline which also does efficient row selection, AQUOMAN has 13% to 23% improvement over MonetDB.

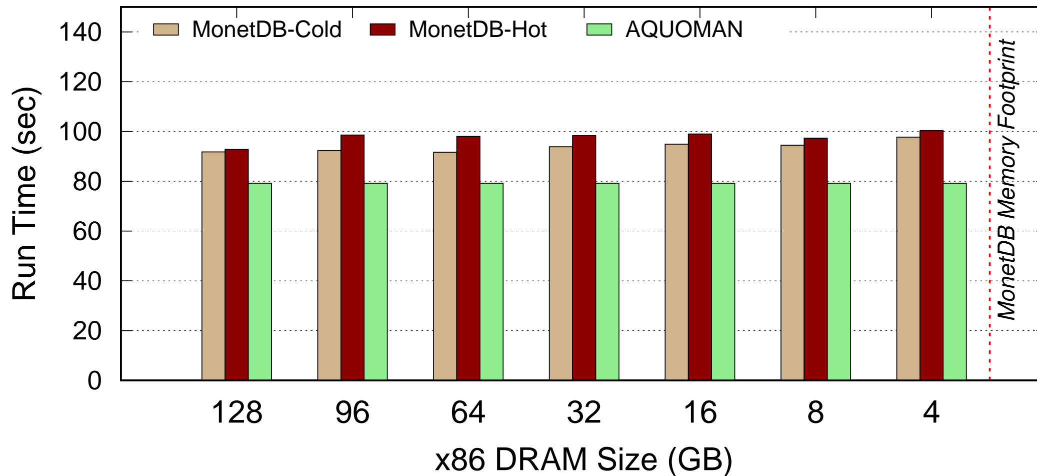


Figure 9-1: Query A: High-selectivity Filter: No Column Reuse

9.3.2 Query B: Query with column reuse

The following query is the Q06 of the TPC-H benchmark.

```
SELECT sum(l_extendedprice * l_quantity) as revenue
FROM lineitem
WHERE l_shipdate >= date '1994-01-01'
  and l_shipdate < date '1994-01-01' + interval '1' year
  and l_discount between 0.06 - 0.01 and 0.06 + 0.01
  and l_quantity < 24;
```

It has the same filter predicate as the previous query, except that the `l_quantity` column is reused for aggregation. Therefore the total input data-set is reduced to 134GB, where the *Row Transformer* and the *Row Selector* do share the `l_quantity` column of 44GB.

Figure 9-2 shows the evaluation result of this query. Due to small memory foot print of this query processing, MonetDB can process query efficiently in full range of DRAM capacities. When the DRAM size for the baseline is 128GB and is near input data-set size, we can see that the baseline has better performance in hot runs than in cold runs, since the system gets page-cache hits. For cases when the DRAM is smaller 128GB, MonetDB begins thrashing the page-cache in hot runs again.

Our current implementation of AQUOMAN does not allow us to effectively reuse the column data between *Row Select* and *Row Transform* since our flash-page buffer on AQUOMAN

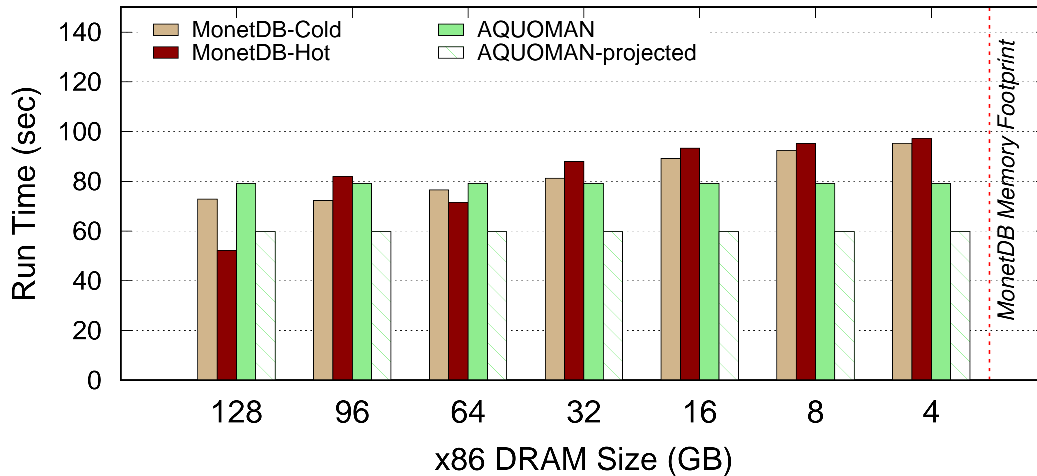


Figure 9-2: Query B: High-selectivity Filter: Column Reuse

has only one read port which matches the speed of the flash controller. The read port is multiplexed by the *Row Select* and *Row Transform*. For processing this query, AQUOMAN has to re-stream the `l_quantity` column from the flash-page buffer at the speed of our flash drive, whereas MonetDB can take advantage of locality by reusing the column data cached in DRAM. AQUOMAN has performance within +/-5% of the baseline when the baseline has DRAM capacity between 32GB to 96GB to effectively cache the reused `l_quantity` column file of 44GB. When the DRAM capacity drops below 32GB, AQUOMAN, without effective column reuse, outperforms the baseline by 1.2X to 1.3X since MonetDB computes selection and aggregation in steps, and DRAM cannot effectively cache the entire `l_quantity` column file. With 128GB DRAM, the performance of the baseline prevails over AQUOMAN by 36% since the page-cache has sufficient capacity to effectively cache the entire input data-set, which allows MonetDB to take advantage of locality opportunities available both between queries and within the query.

If either the data-path size or the clock rate of the flash page buffer inside AQUOMAN doubled, AQUOMAN could make full reuse of column data shared between the *Row Selector* and the *Row Transformer*. Our projected AQUOMAN performance shows that it could have 1.2X to 1.4X improvement over the baseline when MonetDB's DRAM is smaller than 128GB. When the baseline has a 128GB DRAM, AQUOMAN is within -12% of the baseline performance, even though MonetDB is computing with data fetched from a hot page cache at an-order-of-magnitude faster clock speed than our prototype AQUOMAN.

9.3.3 Query C: Aggregation GroupBy

The following query is TPC-H benchmark Q01 without the Order-By operation on the final output table. The Order-By operation is trivial to compute since the final table has only 4 rows of different groups. The total input data set takes 190GB with no shared columns between *Row Select* and *Row Transform*. The filtering predicate of this query has low selectivity which selects 98.6% of the 6 billion rows from the `lineitem` table.

```
SELECT l_returnflag, l_linestatus,
  sum(l_quantity), sum(l_extendedprice),
  sum(l_extendedprice*(1-l_discount)),
  sum(l_extendedprice*(1-l_discount)*(1+l_tax)),
  avg(l_quantity), avg(l_extendedprice), avg(l_discount),
  count(*)
FROM lineitem
WHERE l_shipdate <= date '1998-12-01' - interval '90' day
GROUP BY l_returnflag, l_linestatus;
```

Since this query has such a lower filter rate and need to derive an intermediate table of as many columns as the input *Aggregate Group-By*, the memory footprint on the baseline system is as large as 48GB. Like the previous query, AQUOMAN does not require DRAM for merging *Table Task*s outputs. Indeed only one *Table Task* is needed. Moreover, this query has only 4 groups whose aggregates of 4 columns can be easily stored on AQUOMAN's SRAM.

As shown in the result in Figure 9-3, the performance of the baseline deteriorates dramatically when its DRAM is below 32GB since it starts swapping pages to disk. The baseline cannot finish processing the query within 20 minutes in such scenarios, whereas AQUOMAN dominates over the baseline by producing the result in 86 seconds without the need of DRAM. Even when more than 64GB DRAM is provided to the baseline, MonetDB reserves 48GB of DRAM for computing the query, which leaves little DRAM to cache the input data-set of 190GB. Like in Query A, the baseline in hot runs begins thrashing LRU-based page-cache as explained earlier.

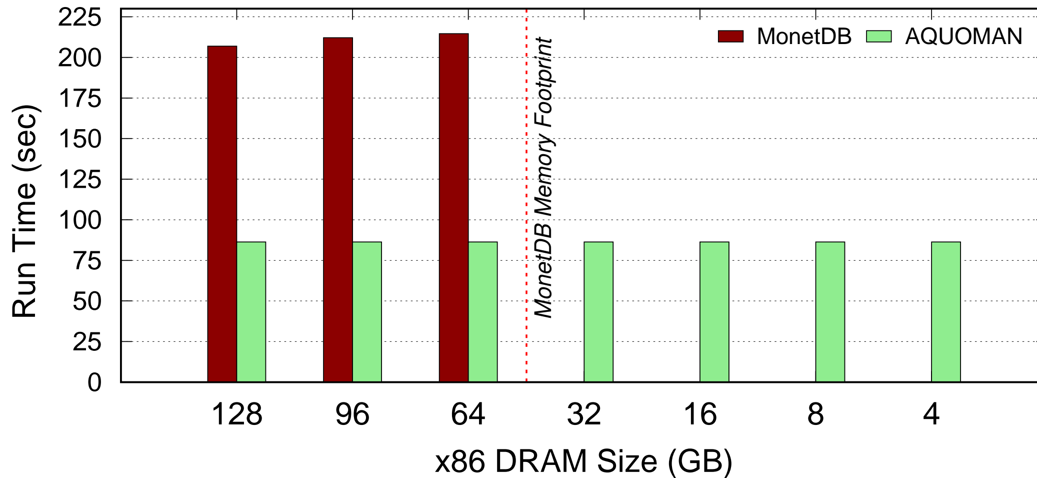


Figure 9-3: Query C: Aggregate-GroupBy Query

When it has sufficient DRAM for intermediate tables, MonetDB can finish processing this query in around 210 seconds. AQUOMAN is 2.5X faster than MonetDB in such scenarios.

9.3.4 Query D: TopK

The following query selects the highest 10 discounted prices whose shipdate is prior to 1998-09-02.

```
SELECT l_extendedprice * (1 - l_discount) as disc_price
FROM  lineitem
WHERE l_shipdate <= date '1998-12-01' - interval '90' day
ORDER BY disc_price desc LIMIT 10;
```

The query uses the same filter predicate as Query C, which yields low selectivity: 98.6% of 6 billion rows. The input data-set is 111GB while the memory footprint of the baseline is around 8GB. The baseline chooses to div-and-conquer this query, where the whole data-set is partitioned. MonetDB computes the top 10 results for each the partition in parallel. Then the top 10 results for each partition are merged from which the final top 10 results are produced. Each top 10 computation is performed using minHeap.

The evaluation result in Figure 9-4 shows that the baseline begins to swap to disk when the memory is below 8GB. With only 4GB of DRAM, the disk swapping becomes predominant such that the baseline cannot finish processing within 20 minutes. Like in

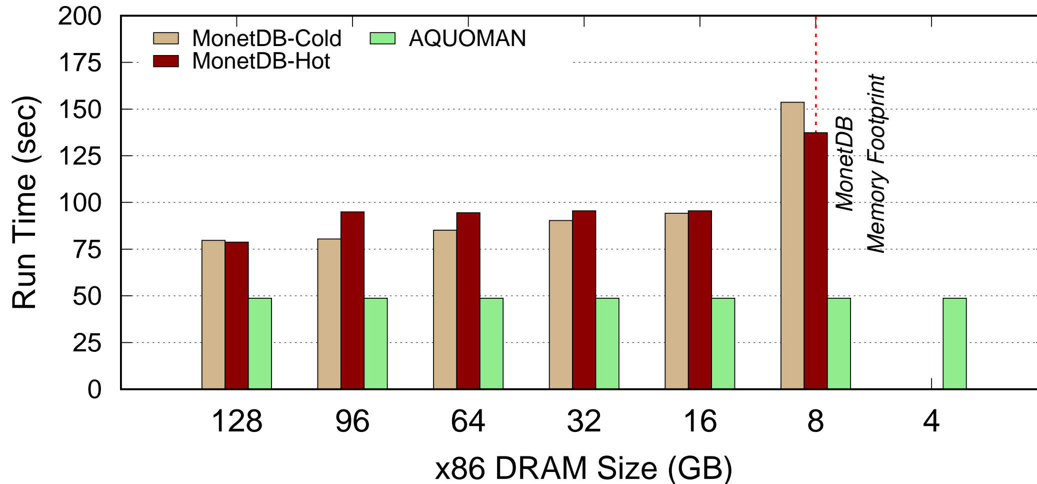


Figure 9-4: Query D: TopK

the previous queries, AQUOMAN's streaming model allows it to perform the computation without DRAM and can produce the result consistently in 49 seconds.

When the baseline has sufficient DRAM, it can finish processing the query between 80 to 95 seconds. It also shows that between 16GB to 96GB, MonetDB starts thrashing the page-cache which makes the hot runs slower than the cold run as explained earlier. In such scenarios AQUOMAN has 1.6X to 2X performance improvement over the baseline.

9.3.5 Discussion on Row Selectivity

Query A and B have low row selectivity. Little data is left to be further processed after row selection. This has been the primary focus of previous in-storage accelerator work [89, 106, 140, 202, 203] for the benefits of PCIe I/O reduction. However x86 processors are good at filtering (mainly arithmetic comparison that can be mapped across threads), thus these type of queries are easily I/O bound and rarely benefit from in-storage acceleration when the PCIe bandwidth is similar to the SSDs' aggregated bandwidth. Query A and B are similar except that Query B reuses column `l_quantity` in row filtering for aggregation. MonetDB can take advantage of locality by reusing the column data cached in DRAM. Query C and D have high row selectivity, so a large amount of data has to be processed by the subsequent operators (e.g. Aggregate-Groupby, TopK, . . .). In those queries, the x86 CPU is stressed before the disk bandwidth becomes a bottleneck. Since AQUOMAN can process subsequent

operators after row selection at line speed, AQUOMAN is faster in such queries.

9.4 Multi-Table Queries

9.4.1 AQUOMAN Multi-table Evaluation Methodology

Currently we cannot fit AQUOMAN and the *Sorter* on a single FPGA, so we are not able to run multi-table queries end-to-end on AQUOMAN. In the design of AQUOMAN, *Table Task*s are executed sequentially. We execute each *Table Task* of a query individually and sums up the execution time of each *Table Task* for the end-to-end query run time. For the *Table Task*s that involve *Sort*, we use a traffic generator as opposed to real data, throttled at the same speed as AQUOMAN's flash card (2.4GB/s). This is because our flash card is incompatible with Xilinx VCU118's newer version of the FMC+ connector.

9.4.2 Query E: Join

```
SELECT sum(l_extendedprice) FROM lineitem, orders
         WHERE l_orderkey = o_orderkey
           and o_orderdate < date '1992-03-15'
           and l_shipdate > date '1992-03-15';
```

Query E is an inner equi-join query which summarizes the revenue of total prices for shipped orders made by 1992-03-15.

Joins of Tables are typically performed on primary keys (sorted) and foreign keys (unsorted). Whenever a column of a table is set as the primary key, MonetDB creates an extra column of *oid*s (object id) materialized to disk, which are sorted row indices of the table. When a column of another table is set as a foreign key, MonetDB creates another column of cross-referencing `oids` to the primary keys of the referred table. MonetDB relies on the `oids` to perform join operators.

AQUOMAN needs three *Table Task*s to compute this query, as shown in Table 9.4.

- *Step 1:* The first *Table Task* works on filtering the `orders` table and produces a sorted column of `oid_o_orderkeys`. AQUOMAN will not need to sort this column since

Table Task Seq.	Operations	Input Sz	Filter Sel.	Output Sz	Execute Tm
0	*Row Select*	17.2GB	5.8%	352MB	7.2s
1	*Row Select* → *Sort_Merge*	68.7GB	99%	1.9GB	28.6s
2	*Sort* → *Aggregate*	45.8GB	NA	NA	19.1s
	Total:	131.6GB			54.9s

Table 9.4: *Table Task*s of Query E

it is already sorted on disk. The intermediate result is 352MB, and is buffered on AQUOMAN's DRAM.

- *Step 2:* The second *Table Task* works on the `lineitem` table by producing sorted 1GB blocks of key-value (KV) pairs of selected <`oid_l_orderkey, oid_l_lineitemkey`>. The `oid_lineitemkey` is the primary key of the `lineitem` table which is needed to select the rows that are aggregated in the third *Table Task*.

The streaming *Sorter* sorts the KV pairs by the `oid_l_orderkey` foreign key. The output stream of the *Sorter* is sorted per 1GB block. AQUOMAN stream-merges each 1GB blocks on the fly with the output of the first *Table Task* coming from AQUOMAN's DRAM. The merged result are KV pairs of the primary keys of the `orders` and the `lineitem` tables: <`oid_o_orderkey, oid_l_lineitemkey`>. It has 127 million entries for a total size of 1.9GB. *Step 3:* The third *Table Task* also works on the `lineitem` table, which reads from DRAM the value part of the key-value pairs <`oid_o_orderkey, oid_l_lineitemkey`>. AQUOMAN will use the *Sorter* again to sort `oid_l_lineitemkey` to generate a row-vector mask, stream out the rows and compute the aggregation. All the intermediate outputs produced by the *Table Task*s fit on AQUOMAN's DRAM, and AQUOMAN can finish the query in 55 seconds. Although we have not fully integrated the *Sorter* with AQUOMAN due to the resource constraint of a single FPGA, we can comfortably extrapolate that AQUOMAN could process Query E in around 55 seconds by consuming the input data size of 131GB at 2.4GB/s.

For Query E, AQUOMAN is 2X-5X faster than all baselines that have enough DRAM (see Figure 9-5). Without enough DRAM the baseline systems start swapping to disk and their performance deteriorate. SSD-A swapping causes bigger penalty since it is near its full capacity, which incurs garbage collection inside the SSD.

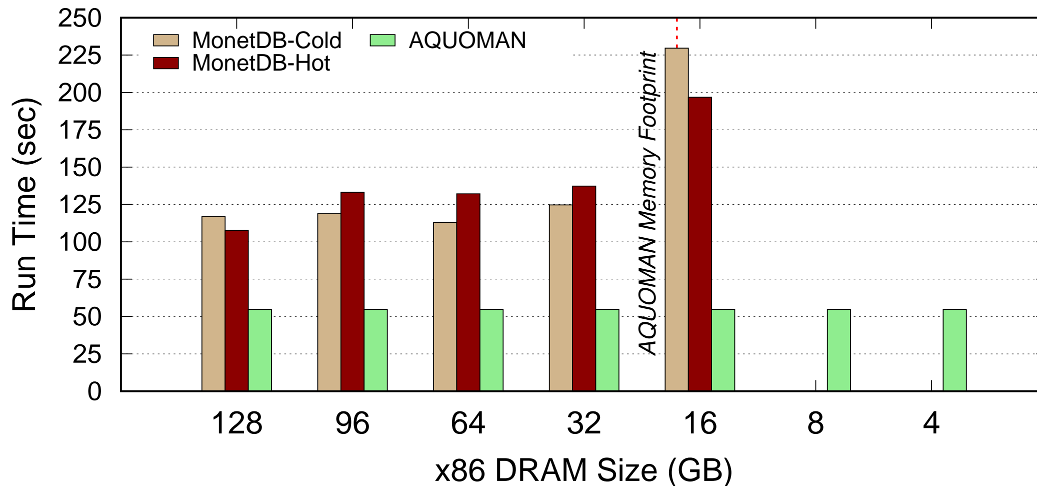


Figure 9-5: Sort-Merge Join(AQUOMAN) vs. Merge Join(MonetDB)

For this particular query, the baseline performs a *merge-join* algorithm because the filtered foreign key/`oids` also happens to be sorted. AQUOMAN does not have this information until it finishes streaming all foreign key `oids`, therefore it cannot do a *merge-join*. Instead AQUOMAN performs a *sort-merge* join. The performance is not affected because AQUOMAN's streaming sorter simply behaves as a long-latency FIFO when the input stream happens to be sorted. AQUOMAN is 2X faster than the baselines even when the baseline performs merge-join. MonetDB has a 16GB memory footprint to process Query E using merge-join.

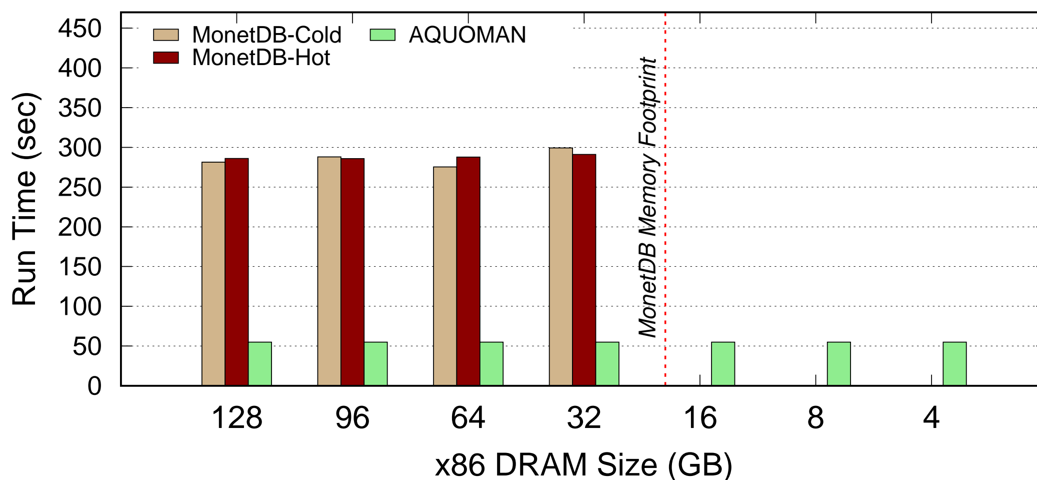


Figure 9-6: Sort-Merge Join(AQUOMAN) vs. Hash Join(MonetDB)

To understand how AQUOMAN competes with a more commonly-used join algorithm,

hash-join, we modified the MonetDB's kernel to make it run hash-join on Query E. Figure 9-5 shows that AQUOMAN is 5.5X faster than all baseline systems. The memory footprint of hash-join is slightly bigger than merge-join since it also needs to keep a hash-table in DRAM while processing the query.

Chapter 10

TPC-H Benchmark with AQUOMAN

AQUOMAN was first implemented on an FPGA, although soon afterwards we discovered several reasons which prevented us from evaluating most TPC-H queries on the FPGA prototype:

- Our FPGA evaluation board has only 4GB of DRAM, which is not big enough to evaluate multi-way joins that generate bigger intermediate tables.
- AQUOMAN with the *Sorter* exceeded the total area of the FPGA in BlueDBM. Our bigger FPGA, VCU118, is not compatible with the FMC port of the custom flash card in BlueDBM.
- A robust regular-expression accelerator, which has been done previously [104], is also needed for string columns but required significantly more implementation effort than this project justified.
- A full-blow compiler is needed to generate *Table Task*s for FPGA evaluation; manual compilation effort is too high for most TPC-H queries. (Investment in such a compiler would be justified only after the efficacy of AQUOMAN has been established.)

In order to evaluate more queries and to properly evaluate the AQUOMAN architecture, we also developed a trace-base AQUOMAN simulator and fully integrated it in the MonetDB's software stack . We validated some of our simulation results on the FPGA proptotype (Section 10.4)

10.1 AQUOMAN Simulator

We implemented a trace-based AQUOMAN simulator which is fully integrated in MonetDB 11.27.9. In MonetDB the software translates the SQL execution plan into a customized middle-layer language, Monet Assembly Language(MAL), which is then later optimized and interpreted [114]. We implemented the AQUOMAN simulator by extending MAL to allow instrumenting traces for AQUOMAN *Table Task*s. The AQUOMAN simulator does not execute *Table Task*s, but executes the original SQL plan expressed in MAL and collects AQUOMAN traces such as flash traffic, AQUOMAN memory footprint, and sorter usage. The AQUOMAN simulator assumes a flash drive of 8KB page access granularity and 2.4GB/s flash read bandwidth, one streaming sorter and one regular expression accelerator with 1MB cache for string heap. The specifications of flash drive and streaming sorter in AQUOMAN simulator are the same with the ones in the AQUOMAN FPGA prototype in Section 4.1.1. We assume as big Row Selector and Row Transformer as needed as their small relative sizes compared to the sorter as shown in Section 4.1.1. When a multi-way join query exceeds AQUOMAN memory size, we assume that the host processes "handed-off" sub-query at the same speed as the baseline solution, which is a conservative assumption.

In the SQL frontend, we modified MonetDB's query planner to identify *Table Task*s in the query execution plan tree, and mark relevant nodes as AQUOMAN nodes which are targets for offloading. We also changed MonetDB's SQL plan to MAL compiler, such that AQUOMAN tracing instrumentation will be automatically inserted on identified *Table Task*s. The total execution time of a query with AQUOMAN simulator is calculated by the sum of AQUOMAN execution time based on the traces and non-AQUOMAN nodes' execution time processed by MonetDB.

10.2 Experiment Setup

10.2.1 Evaluation Data-set

We used the TPC-H synthetic data-set with a scaling factor of 1000, generating **1TB** of tables. We loaded the data-set on MonetDB-11.27.9, whose column files are the inputs for

AQUOMAN.

10.2.2 Baseline Setup

We ran MonetDB (11.27.9) with two setups S and L to represent two different machine sizes (Table 10.1). The baseline used five 1TB Samsung 970 EVO m.2 SSDs capped at 2.4GB/s, to match the bandwidth of the BlueDBM storage device. Such a setup was needed for a fair baseline 1) to mitigate side-effects of garbage collections with over-provisioned capacity and 2) to provide 2.4GB/s average access bandwidth unavailable in a single off-the-shelf SSD. When MonetDB run out of DRAM for intermediate tables, it can still process queries effectively by using its own disk-swap management, which exploits fast sequential SSD writes.

MonetDB does not implement a page buffer pool for caching hot pages, instead it relies on Linux's LRU-based page cache to take advantage of page locality. For 1TB dataset, we observed that Linux's page cache on a 128GB DRAM is ineffective for TPC-H queries. In fact, for MonetDB hot runs are slightly slower than cold runs, even though both runs experience a similar level of page-cache misses. We hypothesize that the cost of finding misses in a fully populated page-cache is larger because the operating system needs to traverse a bigger page-cache index structure to find nothing. Therefore our evaluations assume cold page cache.

x86 Setup	HW Threads	x86 DRAM
S (Small)	4 Threads	16GB
L (Large)	32 Threads	128GB

AQUOMAN Setup	DRAM
AQUOMAN	40GB
AQUOMAN16	16GB

Table 10.1: x86 Host and AQUOMAN Disk Setup

10.2.3 AQUOMAN Setup

All TPC-H queries are evaluated on the AQUOMAN simulator, which has two setups: AQUOMAN with 40GB memory and AQUOMAN16 with 16GB memory (Table 10.1). The AQUOMAN implementation on FPGA could be used for evaluating only a few TPC-H queries

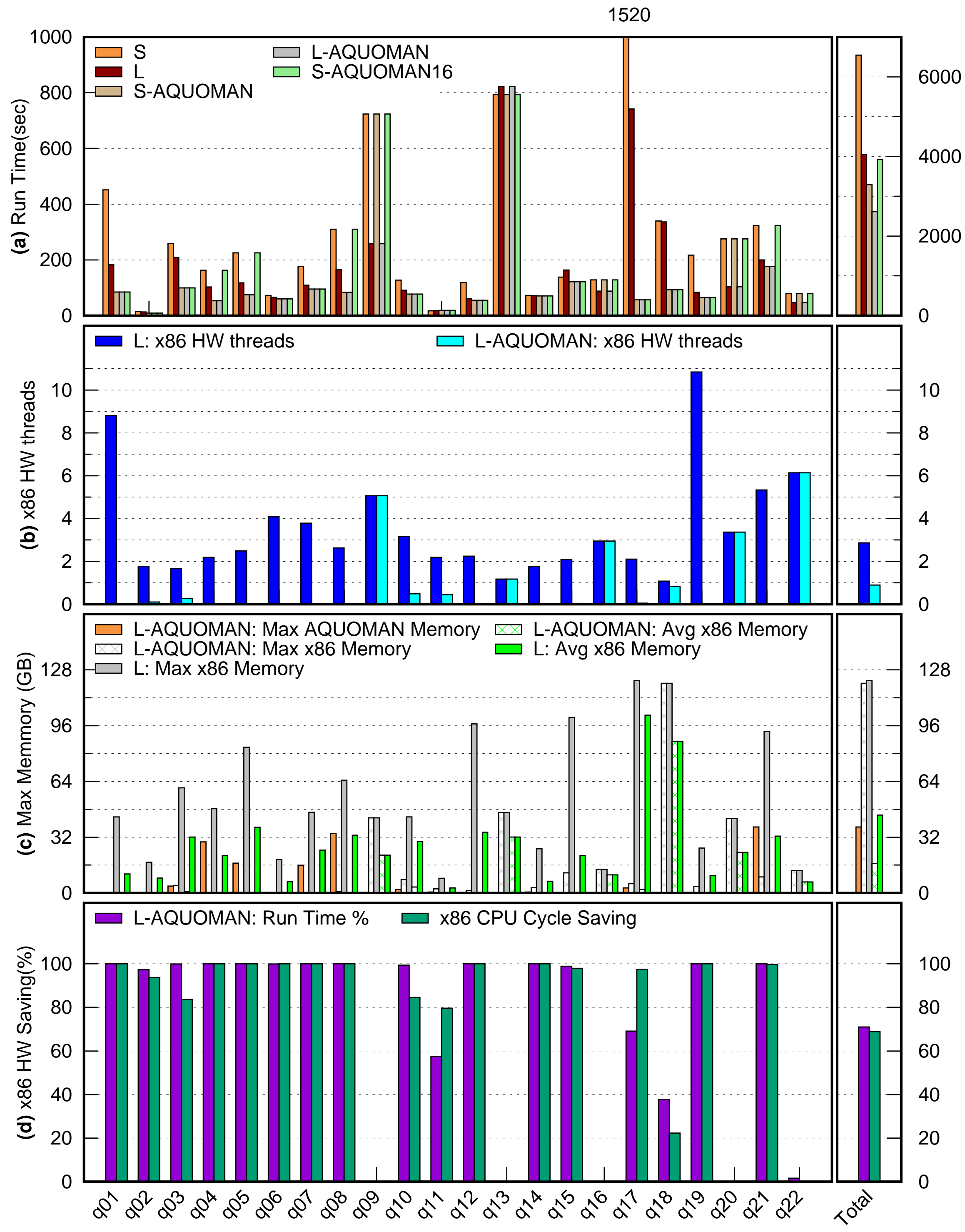


Figure 10-1: TPC-H SF-1000 AQUOMAN Performance Profiling

because of the reasons discussed earlier (Section 10). However, the FPGA implementation was very useful to validate the AQUOMAN simulator (Section 10.4).

10.3 AQUOMAN TPC-H Evaluation

We compare the performance of MonetDB running on a system with ordinary SSDs with a system where the ordinary SSDs are replaced by AQUOMAN SSDs. For an extensive coverage of the design space, two host machines S, L (Figure 10.1) are used, each with and without AQUOMAN disks. We also paired the small host System S with AQUOMAN16 disk, which has 16GB of in-storage DRAM.

We first examine the runtime of each query, including the breakdown of the time spent on AQUOMAN and the host. We then perform a similar analysis for the memory footprint.

10.3.1 Run Time

The queries run time for the different systems are presented in Figure 10-1(a), while the fraction of processing time each query spent on AQUOMAN for system (L) is shown in Figure 10-1(d) . Let us first focus on L and L-AQUOMAN, that we study in more detail. On average, 71% of the CPU time can be saved by AQUOMAN when it is added to System L. Note that AQUOMAN actually speeds up many queries, on average a 1.5X-2X speed-up over the baseline in Figure 10-1(a). Still there are some outliers, queries (17,18) show up to 13X speed-ups for System L, while others show none. It is important to realize that AQUOMAN cannot speed up a query if it is IO bound in the baseline system. In such cases it can only save host resources. Note that a processing time dominated by AQUOMAN processing does not necessarily means a high speed-up. For example, we found that two queries (6,14) can be almost completely off-loaded to AQUOMAN but show little speedup because they are disk-bound on the baseline systems.

For 14 out of the 22 queries are off-loaded to AQUOMAN nearly 100% of the time. Even when AQUOMAN can only do a part of the query, its resulting benefits can be significant. For example, the runtimes of Q17 and Q18 decrease significantly because the part that is

off-loaded happens to execute sequentially on the host, effectively using only one hardware thread.

The reasons for queries to be suspended early were discussed in Section 8.5. As we said earlier queries (17,18,22,11) corresponds to cases with an early *Aggregate Group-By* node in the execution plan. All of them except Q22 do enough processing on AQUOMAN to show speedups. Queries (9,13,16,20) represent the cases where the size of the string heap does not fit in AQUOMAN and so have to be completely handled by the host.

Figure 10-1(b) shows the benefits of AQUOMAN in terms of changes in x86 HW threads. System L requires a maximum of 11 threads to pair with a single SSD's bandwidth when executing computational intensive queries such as q14. On average System L needs 3 HW threads per SSD for the entire SSD benchmark. Please note although System S has 4 HW threads, some query execution time on System S would drop because of insufficient DRAM for storing intermediate results (See Figure 10-1(c)). When System L is paired with AQUOMAN, the maximum x86 HW threads can be reduced to 4 and the average can be reduced to 1.1 Overall when a host machine replaces its SSDs with an AQUOMAN disk, it can save on an average 71% of the CPU time for TPC-H queries running on System L.

10.3.2 Memory Footprint

Figure 10-1(c) shows the maximum and average memory resident set size (RSS) of AQUOMAN and the system-L baseline, respectively. When a query is adequately offloaded, AQUOMAN reduces host memory footprint significantly except when it has to aggregate on a huge number of groups in the host as for Q18 (See Section 8.5). AQUOMAN has 20~128GB smaller memory footprint than the baseline even when most of the query is processed by AQUOMAN. The memory saving is primarily because of the streaming model of *Table Task*s and only keeping row IDs in the DRAM. The maximum memory requirement for the TPC-H benchmark by AQUOMAN on 1TB data-set is 40GB. In fact when equipped with 16GB DRAM, only 4 queries (4,5,8,21) are affected and AQUOMAN can offload 12 of 22 TPC-H queries profitably. We also note that while AQUOMAN reduces the average DRAM used significantly (by a factor 3), the maximum DRAM needed is left almost unchanged. Indeed

an important part of Q18 has to be processed by the host, and it requires almost 128GB of DRAM.

10.3.3 Advantages of AQUOMAN

In the previous section we have shown significant hardware threads and memory savings. In this section we propose a way to visualize the benefits from those savings (70% of CPU and 60% of the average memory). As a first approximation, Figure 10-1(a) shows that running the entire TPC-H benchmark on a 32-cores machine with 128GB DRAM (System L) is on average 1.6X as fast as running the same benchmark of on a 4-core machine with 16GB DRAM (system S). However, replacing the SSD of the small machine by an AQUOMAN augmented SSD (S-AQUOMAN16) bridges that gap completely!

This comparison does not evaluate the opportunity for the system to run many queries in parallel, which may show different results because of inter-query data locality and parallelization. Evaluating such a parallel system requires a very different setup than what we have presented in this paper.

10.4 Validating simulation results on FPGA

The AQUOMAN FPGA prototype has the key AQUOMAN components: *Row Selector*, *Row Transformer*, and *SQL Swissknife* with a high-performance *Sorter* but is limited by 4 GB of DRAM. We evaluated a subset of TPC-H queries using the AQUOMAN FPGA prototype to validate some of our simulation results.

We picked two classes of TPC-H queries and hand-coded them to *Table Task*s to execute on the FPGA prototype. The first type of queries (1,6) have no join operations. Those queries are evaluated end-to-end and produce the same query results as the MonetDB software. The second type of queries (3,10) are multi-way join queries but need less than 4GB AQUOMAN DRAM. We used the same evaluation methodology described in Section 9.4.1 for join query evaluations on our FPGA.

For each query, we compared the run time and memory usage of the FPGA prototype with those of AQUOMAN simulator (Figure 10-2). We can see the FPGA prototype has

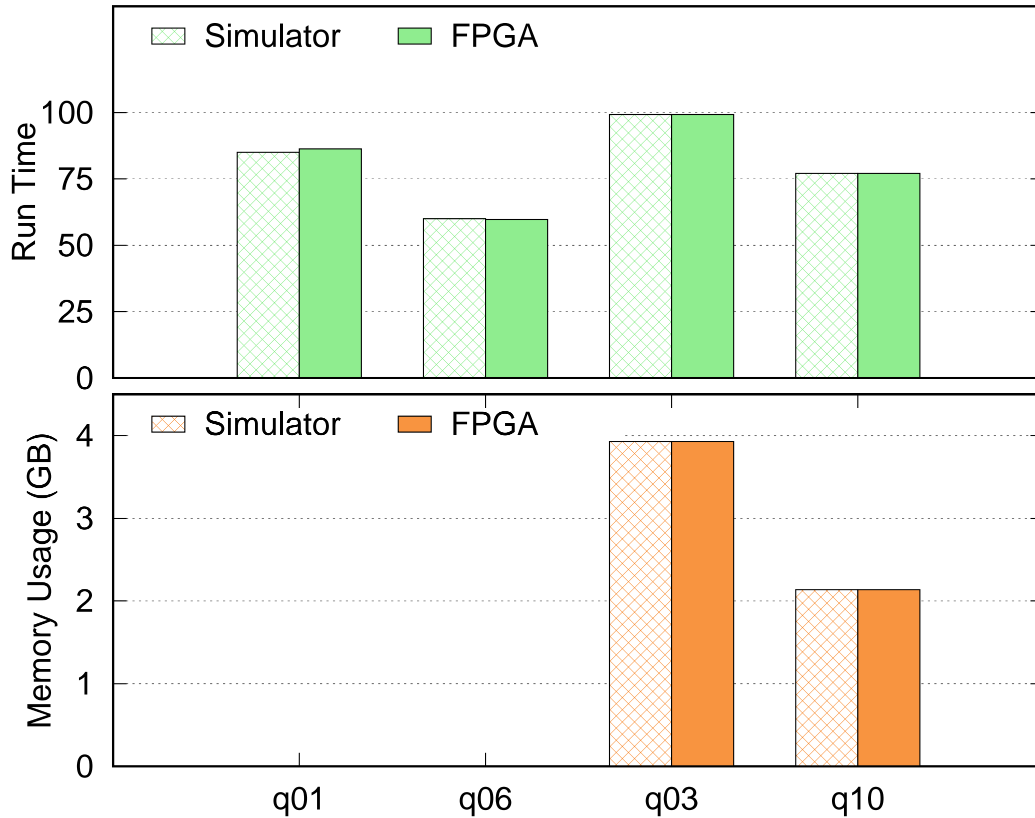


Figure 10-2: TPC-H queries on FPGA prototype

similar run times and the same memory usage as the AQUOMAN simulator, which validates the basic performance modeling of AQUOMAN.

Compared to FCAccel [202], our FPGA evaluation used 10X the dataset size and evaluated more queries including fully-offloaded joins. Therefore we cannot provide direct query run-time comparison, but we can compare in terms of rows/sec with FCAccel's evaluation. AQUOMAN's FPGA performance is competitive to that of FCAccel. For the straightforward filter-and-aggregate with high selectivity (Q6), AQUOMAN has similar throughput (100.5M rows/s vs. 111M rows/s). When a query has low selectivity and requires more computation, such as row transform and *Aggregate Group-By* in Q1, AQUOMAN is 2.5X better than FCAccel (69M rows/s vs. 27M rows/s). This is thanks to AQUOMAN 's systolic-array design for highly-pipelined row transformation (Section 8.2), while FCAccel uses on multi-cycle logic designs.

10.5 Part II Conclusion

We have presented AQUOMAN, an end-to-end DBMS system solution for in-storage analytical SQL query acceleration. AQUOMAN aggressively pushes the idea of "in-storage computing" by offloading most of the query processing, including multi-way *Join*s, for terabyte data-sets. AQUOMAN is based on a novel stream-oriented microarchitecture to execute static dataflow graphs of SQL operators organized as *Table Task*s. We have built a prototype of AQUOMAN using a Xilinx VCU108 FPGA development board, and shown that it computes *Table Task*s at a 4GB/s, saturating the flash-drive bandwidth. (For power, cost and area reasons, a commercially viable version of AQUOMAN will have to be implemented using ASICs).

We have run experiments to show a clear-winning case for AQUOMAN over existing general purpose servers when page-cache is ineffective. When data-set size is just too big or temporal locality of queries are limited, general-purpose machine can struggle to provide enough DRAM capacity to offer effective page-caching of database tables. In such cases, AQUOMAN can provide compelling performance than general-purpose solutions because both solutions are both disk-IO bound and AQUOMAN has superior query processing efficiency because of specialization. We have shown AQUOMAN can have a 1.2X-13.0X single-query performance improvement when MonetDB's page cache becomes ineffective. We have also shown that an AQUOMAN instance can free up 2~20 HW threads and 20~128GB DRAM needed by a pure software solution to saturate a single SSD.

One way to think of the benefits provided by offloading queries to AQUOMAN is to imagine running SQL queries on a one-terabyte TPC-H benchmark data-set on two systems: MonetDB running on a 4-core, 16GB-DRAM machine with AQUOMAN-augmented SSDs and MonetDB running on a 32-core, 128GB-DRAM machine with standard SSDs. Overall, if we run TPCH queries sequentially and assume no reuse of page-cache by different queries, the two system provide the same performance.

Future work needs to be done to determine if AQUOMAN can win over general-purpose solution when the latter has enough DRAM for caching reused pages. It is conceivable AQUOMAN can still win if there are enough aggregate disk bandwidth allocated for AQUO-

MAN, because database queries mostly results in sequential scans of rows of tables and sequential flash drive reads are highly efficient. Such work on AQUOMAN requires (1) an experimental setup to evaluate parallel execution of queries and (2) distributed execution of queries whose data is spread over multiple AQUOMAN SSDs.

Chapter 11

Conclusion and Future Work

Existing computing systems process increasingly large amounts of data. Data is key for many modern workloads. Important workloads, such as machine learning/artificial intelligence, bioinformatics, graph analytics, databases, video analytics, are data-intensive and needs the underling computing systems to process a vast amount of data that is continuously being generated today. Modern computing systems are typically processor-centric, where computing units are interconnected via highly sophisticated hierarchies of memory subsystems, networks and storage devices. Running large-scale data processing on such systems requires bringing data piece-wise over the memory/network-hierarchy into the processing logic pipeline to perform computation, which exists both in the micro-level (single-node machine) and macro-level (server clusters). Such processor-centric designs leads to large amounts of data movement across the entire system, degrading performance and consuming the majority of machine energy [61]. With the exponential increase of data generated by applications, future worloads can quickly overwhelm the storage capacity(cache/DRAM), communication capability(processor interconnect/memory bus/network), and computational capability(general-purpose processors) of modern machines we design today. Meanwhile caches and interconnects of a modern processor can easily take the majority of processor area(e.g. 80-95%), and their efficacy is quickly diminishes with decreasing data locality/reuse for large data sets. I believe that the challenges of future computing systems is to address the performance, efficiency and scalability issues related to the data-movement bottlenecks for data-intensive applications.

My research goal was to build future data-centric computing systems which can offer high programmability, high energy-efficiency and high performance for data-intensive workloads. Fundamentally such data-centric computing systems should minimize data-movement by enabling computational capability near where data resides. Two types of memory technologies can potentially offer high efficiency as the memory substrate for data-centric computing for large data-sets. The first is flash-centric computing systems which performs computation in the secondary storage, such as SSDs. This thesis has shown such systems can easily address several terabytes of data using a single node, and can offer competitive performance with extreme power efficiency [124, 129, 207, 208]. And I plan to build on my ongoing flash-centric work in the short term.

The second idea is near-memory computing, which is driven by the 3D-stacked memory technology, such as hybrid memory cubes (HBM). 3D stacked-memory technology allows logic/memory layers to be vertically stacked and exposes massive memory channels across vertical layers. Near-memory computing can offer high-performance for a wide class data-intensive applications that has hundreds of gigabytes dataset and can fit in memory (E.g. in-memory database, graph-pattern mining). I am also interested in designing and building memory-centric systems using latest memory technologies in the longer term. This thesis laid out some initial steps towards achieving these goals, and I am excited to use my skills to continue solving more interesting problems in this area.

11.1 Short-term Future Work

I plan to build upon my ongoing flash-centric hardware accelerators research as follows:

11.1.1 Distributed In-storage SQL analytic Offloading

The previous study of AQUOMAN focused on a single instance of such an offloading hardware accelerator. The AQUOMAN architecture assumes local access to flash chips for input data and thus, required that the data accessed by a query reside on a single SSD. I plan to build a a distributed version of AQUOMANs, i.e. AQUAFarm, to process much bigger datasets and to provide query-processing scalability on hundreds of terabytes or

petabytes of datasets. I plan to use map-and-reduce compute paradigm to allow many offloading accelerators cooperate on a single query. The scalability of executing Table Tasks on AQUAFarm is highly dependent on the amount of data that needs to be shuffled over the network. To reduce network traffic, data transmission can be efficiently compressed using delta encoding since the Table Task outputs are sorted. To make the SQL queries run transparently on AQUAFarm, I plan to integrate AQUAFarm into Presto [188], Facebook's open-source distributed SQL engine for big-data SQL analtyics.

11.1.2 Graph-Pattern Mining using Accelerated Flash Storage

Prior work in graph-pattern mining (GPM) accelerators [74,132,209] assumes graph datasets are DRAM resident, which limits the size of processing of large graphs to 10-100 gigabytes. In order to mine sub-graphs in terabytes of graph datasets without using a large distributed server cluster with enough aggregate DRAM for storing input graphs, I plan to build on our previous work, FlexMiner [74], to process GPM in storage. One of challenges in processing GPM using flash storage is that while extending a sub-graph, flash access granularity leads to significant read amplification. I plan to explore a memory subsystem which sorts and merges flash read requests from a massive amount of parallel mining engines, to make most efficient use of secondary storage I/O.

11.1.3 Machine Learning Accelerators

Many important applications now arise in the area of Artificial Intelligence, where their needs are poorly met by current storage systems. With the rapid growth of data for a wide range of AI applications, this space holds exciting opportunities for impactful research. I plan to expand my research into accelerating machine learning algorithms which exhibits highly intensive and irregular data access patterns, such as Graph Neural Networks(GNNs).

11.2 Long-Term Future Work

My long term goals are 1) to make programming flash-centric computing easier, and 2) to design and build memory-centric machines using emerging memory technologies.

11.2.1 Make programming flash-centric computing easier

One of my longer-term goals is to make programming hardware accelerators for flash centric computing easier for a wider audiences across different disciplines. One way to make the use of hardware accelerators more widespread is designing good system integration with the software package for the end-users, just like what I did in AQUOMAN [207]. In this case, users run their applications transparently without any knowledge of hardware accelerators. The second way is to provide a programming interface for hardware accelerators by designing a domain-specific language (DSL). This allows the domain experts to upload programming parameters to accelerators for each query, like what we did in Flexminer [74]. Both methods allows hardware accelerators to be fixed functions in ASIC, and I am interested in designing and building flash controller SoCs with hardware accelerators, as well as provide end-to-end software package for them.

The third way is to design and develop a programming model and platform for general-purpose programming on *reconfigurable* hardware accelerators, just like how Hadoop and Spark made distributed computing using map-and-reduce paradigm easier. In this scenario, programmers need to define their own hardware modules using RTL, Bluespec, HLS or others, but needs to obey a certain execution model, such as map-and-reduce. I am interested in using all three methods to make programming flash-centric hardware accelerators easier to widespread their dividends in performance and efficiency.

11.2.2 Memory-centric computing with emerging memory technology

Emerging storage technologies, such as PRAM, MRAM, and Intel X-points, challenge fundamental assumptions in systems design. For example, the storage bandwidth scaling has dramatically exceeded DRAM bandwidth scaling to CPUs [95, 165], and the bottlenecks of many data-intensive workloads has shifted from I/O bandwidth to CPU with newer

storage devices [150]. At the same time, latest silicon manufacturing has allowed integrating layers of silicon logic and memory(e.g. Hybrid Memory Cubes) on a 3D stack, which opens up opportunities and challenges for hardware architectures for emerging applications. The push of 3D-stacked memory and emerging NVM has opened up new frontiers of near-memory computing, where processing units can be vertically stacked with gigabytes of memory and have tens of terabytes per second bandwidth. To offload data-intensive computation efficiently near memory, I am interested in studying different forms of near-memory computing, such as fixed accelerators, FPGAs and light-weight processor arrays, which have different trade-offs of generality and performance.

Meanwhile general-purpose processors can still access 3D-stacked memory via a traditional memory bus. This opens up a new paradigm on how to design *memory-centric* machines which can efficiently and easily bifurcate computation in processors and near-memory logic. One can imagine that cache-friendly computation is more efficient on processors, while data intensive workload with less locality opportunity can benefit from computing near memory. In memory-centric computing conventional processors can cooperate on computation with near-memory logic in a much more fine-grained manner. Therefore, new system challenges such as coherence and virtual memory are likely to arise in memory-centric computing. I am excited to investigate such system challenges to enable ease and efficient migrating processor-centric programs to memory-centric computing and pursue building real SoC systems of such memory-centric computing machines in the future.

Bibliography

[1] 10 Things You Might Not Know About Using S3. https://www.sumologic.com/insight/10-things-might-not-know-using-s3/. accessed: 2021-02-27.

[2] 256GB DDR4 World's First 256GB DDR4 RDIMM. https://samsungsemiconductor-us.com/256/index.html. accessed: 2021-02-27.

[3] 400GbE Finally Hits the Mainstream. https://www.delltechnologies.com/en-us/blog/400gbe-finally-hits-the-mainstream/, Feb.

[4] A Hash Function for Hash Table Lookup. http://burtleburtle.net/bob/hash/doobs.html. accessed: 2021-01-30.

[5] Akamai Online Retail Performance Report: Milliseconds Are Critical. https://www.akamai.com/us/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp. accessed: 2021-02-27.

[6] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. accessed: 2021-02-27.

[7] Amazon Redshift - Data Warehouse Solution - AWS. https://aws.amazon.com/redshift. accessed: 2021-01-30.

[8] Amazon Web Services (AWS) - Cloud Computing Services. https://aws.amazon.com. accessed: 2021-01-30.

[9] AMD will support DDR5 and PCIe 5.0 in 2022, but Intel has DDR5 first. https://www.tweaktown.com/news/72104/amd-will-support-ddr5-and-pcie-5-0-in-2022-but-intel-has-first/index.html, Apr.

[10] Apache Parquet. http://parquet.apache.org/. accessed: 2021-02-27.

[11] Big Data Analytics On-Premises, in the Cloud, or on Hadoop | Vertica. https://vertica.com/. accessed: 2020-04-07.

[12] Bluespec Inc. http://www.bluespec.com.

[13] Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057, note=accessed: 2021-02-27.

[14] Dell EMC SCv3000 Series Storage Arrays. https://www.dell.com/en-us/work/shop/productdetailstxn/storage-scv3020. accessed: 2021-02-27.

[15] Google Flu Thrends. https://en.wikipedia.org/wiki/Google_Flu_Trends. accessed: 2021-02-27.

[16] HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. accessed: 2021-02-27.

[17] In Modern Datacenters, the Latency Tail Wags the Network Dog. https://www.nextplatform.com/2018/03/27/in-modern-datacenters-the-latency-tail-wags-the-network-dog/. accessed: 2021-02-27.

[18] Intel And Micron Jointly Unveil Disruptive, Game-Changing 3D XPoint Memory, 1000x Faster Than NAND. http://hothardware.com/news/intel-and-micron-jointly-drop-disruptive-game-changing-3d-xpoint-cross-point-memory-1000x-faster-than-nand#KyJQDMZWPBQh6KzU.99. accessed: 2021-02-27.

[19] Intel® Optane™ SSD 905P Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-905p-series/905p-960gb-2-5inch.html. accessed: 2021-02-27.

[20] Memblaze's PBlaze5 X26: Toshiba's XL-Flash-Based Ultra-Low Latency SSD. https://www.anandtech.com/show/14708/memblazes-pblaze5-x26-toshibas-xl-flash-based-ultralow-latency-ssd. accessed: 2021-01-30.

[21] Micron Advances Persistent Memory with 32GB NVDIMM. https://investors.micron.com/news-releases/news-release-details/micron-advances-persistent-memory-32gb-nvdimm. accessed: 2021-02-27.

[22] Microsoft Azure Cloud Computing Platform & Services. https://azure.microsoft.com. accessed: 2021-01-30.

[23] Ncbi Genome Database. http://www.ncbi.nlm.nih.gov/genome/. accessed: 2021-02-27.

[24] Netflix EVCache. http://techblog.netflix.com/2012/01/ephemeral-volatile-caching-in-cloud.html. accessed: 2021-01-30.

[25] Nvidia GeForce GTX 780 Specifications. https://www.nvidia.com/en-us/geforce/graphics-cards/geforce-gtx-780/specifications/. accessed: 2021-01-30.

[26] Presto on Amazon EMR - Amazon Web Services (AWS). https://aws.amazon.com/emr/details/presto/. accessed: 2020-04-07.

[27] Redis. http://redis.io.

[28] S3 Select and Glacier Select – Retrieving Subsets of Objects. https://aws.amazon.com/blogs/aws/s3-glacier-select/. accessed: 2020-04-07.

[29] Samsung 870 EVO. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/870-evo-sata-2-5-ssd-4tb-mz-77e4t0b-am/. accessed: 2021-02-27.

[30] Samsung 950 PRO. http://goo.gl/DCwQpd.

[31] Samsung 980 Pro. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-2tb-mz-v8p2t0b-am/. accessed: 2021-02-27.

[32] Samsung 980 Pro M.2 NVMe SSD Review: Redefining Gen4 Performance. https://www.tomshardware.com/reviews/samsung-980-pro-m-2-nvme-ssd-review/2. accessed: 2021-02-27.

[33] SAMSUNG Memory 256GB (2 x 128GB) 288-Pin DDR4 SDRAM ECC Registered DDR4 2933 (PC4 23400) Server Memory Model M393AAG40M3B-CYFC0. https://www.newegg.com/samsung-256gb-288-pin-ddr4-sdram/p/1X5-000A-00PM8. accessed: 2021-02-27.

[34] Seagate announces 64TB NVMe SSD, Updated Nytro NVMe and SAS Lineup at FMS 2017. https://www.custompcreview.com/news/seagate-announces-64tb-nvme-ssd-updated-nytro-nvme-sas-lineup-fms-2017/. accessed: 2021-01-30.

[35] SmartSSD Compuational Storage Drive. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html. accessed: 2021-02-27.

[36] SmartSSD Computational Storage Drive. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html, Apr.

[37] Storage Performance Development Kit. https://spdk.io/. accessed: 2021-02-27.

[38] The column-store pioneer | MonetDB. https://monetdb.org/home. accessed: 2020-04-07.

[39] Uber's big data platform: 100+ petabytes with minute latency. https://eng.uber.com/uber-big-data-platform/. accessed: 2020-04-07.

[40] Virtex Ultrascale+. https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html. accessed: 2021-02-27.

[41] Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/vcu118.html. accessed: 2021-02-27.

[42] Scaling memcached at Facebook. https://www.facebook.com/note.php?note_id=39391378919, Dec 2008. accessed: 2021-02-27.

[43] AWS announces AQUA for Amazon Redshift (preview). https://aws.amazon.com/about-aws/whats-new/2020/12/aws-announces-aqua-for-amazon-redshift-preview/, Dec 2020. accessed: 2021-02-27.

[44] A. Agarwal, M. C. Ng, and Arvind. A comparative evaluation of high-level hardware synthesis using reed–solomon decoder. *IEEE Embedded Systems Letters*, 2(3):72–76, Sep. 2010.

[45] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and Eric Sedlar. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 245–258, New York, NY, USA, 2017. ACM.

[46] Shahriar Akter and Samuel Fosso Wamba. Big data analytics in e-commerce: a systematic review and agenda for future research. *Electronic Markets*, 26(2):173–194, 2016.

[47] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[48] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[49] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 44–54, New York, NY, USA, 2006. Association for Computing Machinery.

[50] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 211–224, Berkeley, CA, USA, 2011. USENIX Association.

[51] Cagri Balkesen, Nitin Kunal, Georgios Giannikis, Pit Fender, Seema Sundara, Felix Schmidt, Jarod Wen, Sandeep Agrawal, Arun Raghavan, Venkatanathan Varadarajan, Anand Viswanathan, Balakrishnan Chandrasekaran, Sam Idicula, Nipun Agarwal, and Eric Sedlar. Rapid: In-memory analytical query processing engine with extreme

performance per watt. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1407–1419, New York, NY, USA, 2018. ACM.

[52] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, Renton, WA, July 2019. USENIX Association.

[53] Banerjee, Hsiao, and Kannan. Dbc—a database computer for very large databases. *IEEE Transactions on Computers*, C-28(6):414–429, June 1979.

[54] Sumita Barahmand and Shahram Ghandeharizadeh. BG: A benchmark to evaluate interactive social networking actions. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[55] Jeff Barr. The Floodgates Are Open – Increased Network Bandwidth for EC2 Instances. `https://aws.amazon.com/blogs/aws/the-floodgates-are-open-increased-network-bandwidth-for-ec2-instances/`. accessed: 2021-02-27.

[56] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, July 2012.

[57] Mateusz Berezecki, Eitan. Frachtenberg, Mike. Paleczny, and Kenneth Steele. Many-core key-value store. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[58] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching – dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, October 2018. USENIX Association.

[59] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10gbps line-rate key-value stores with fpgas, 2013.

[60] Michaela Blott, Ling Liu, Kimon Karras, and Kees Vissers. Scaling out to a single-node 80gbps memcached server with 40terabytes of memory. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara, CA, July 2015. USENIX Association.

[61] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy

Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 316–331, New York, NY, USA, 2018. Association for Computing Machinery.

[62] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.

[63] R. H. Canaday, R. D. Harrison, E. L. Ivie, J. L. Ryder, and L. A. Wehr. A back-end computer for data base management. *Commun. ACM*, 17(10):575–582, October 1974.

[64] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 63–72, New York, NY, USA, 2019. Association for Computing Machinery.

[65] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 151–160, New York, NY, USA, 2014. ACM.

[66] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, 2010.

[67] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 7:1–7:13, Piscataway, NJ, USA, 2016. IEEE Press.

[68] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 387–400, New York, NY, USA, 2012. Association for Computing Machinery.

[69] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. An fpga memcached appliance. In *Proceedings*

of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, pages 245–254, New York, NY, USA, 2013. ACM.

[70] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, page 181–192, New York, NY, USA, 2009. Association for Computing Machinery.

[71] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, page 266–277, USA, 2011. IEEE Computer Society.

[72] R. Chen and V. K. Prasanna. Accelerating equi-join on a cpu-fpga heterogeneous platform. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 212–219, May 2016.

[73] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 269–284, New York, NY, USA, 2014. Association for Computing Machinery.

[74] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21.

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

[76] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, page 609–622, USA, 2014. IEEE Computer Society.

[77] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, page 249–261, USA, 2007. IEEE Computer Society.

[78] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM.

[79] Eric S. Chung, John D. Davis, and Jaewon Lee. Linqits: Big data on little clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 261–272, New York, NY, USA, 2013. ACM.

[80] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. System software for flash memory: A survey. In *Proceedings of the 2006 International Conference on Embedded and Ubiquitous Computing*, EUC'06, pages 394–404, Berlin, Heidelberg, 2006. Springer-Verlag.

[81] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226. ACM, 2016.

[82] Arup De, Maya Gokhale, Rajesh Gupta, and Steven Swanson. Minerva: Accelerating data analysis in next-generation ssds. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '13, pages 9–16, Washington, DC, USA, 2013. IEEE Computer Society.

[83] Jeffrey Dean. Designs, Lessons and Advice from Building Large Distributed Systems. Keynote Speech at The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, October 2009.

[84] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[85] Biplob K. Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *PVLDB*, 3(2):1414–1425, 2010.

[86] David J. DeWitt. Direct - a multiprocessor organization for supporting relational data base management systems. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, ISCA '78, pages 182–189, New York, NY, USA, 1978. ACM.

[87] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[88] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, June 1984.

[89] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[90] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[91] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The mondrian data engine. *SIGARCH Comput. Archit. News*, 45(2):639–651, June 2017.

[92] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 92–104, New York, NY, USA, 2015. Association for Computing Machinery.

[93] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[94] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

[95] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan. phoenix: Memory speed hpc i/o with nvm. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*.

[96] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, March 2017. USENIX Association.

[97] J. Fowers, J. Kim, D. Burger, and S. Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 52–59, May 2015.

[98] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th*

Annual International Symposium on Computer Architecture, ISCA '18, page 1–14. IEEE Press, 2018.

[99] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[100] Eric S Fukuda, Hiroaki Inoue, Takashi Takenaka, Dahoo Kim, Tsunaki Sadahisa, Tetsuya Asai, and Masato Motomura. Caching memcached at reconfigurable network interface. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.

[101] Fusion IO. using membrain as a flash-based cach. `http://www.fusionio.com/blog/scale-smart-with-schooner`, December 2011.

[102] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936, Oct 2020.

[103] Ahmad Ghandour. Big data driven e-commerce architecture. *International Journal of Economics, Commerce and Management*, 3(5):940–947, 2015.

[104] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. Hare: Hardware accelerator for regular expressions. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[105] Rick Greenwald, Mans Bhuller, Robert Stackowiak, and Maqsood Alam. *Achieving extreme performance with Oracle Exadata*. McGraw-Hill, 2011.

[106] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press.

[107] Susan Gunelius. The Data Explosion in 2014 Minute by Minute – Infographic. `http://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic/`, July 2014.

[108] Robert J. Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh Asaad, and Balakrishna Iyer. Accelerating join operation for relational databases with fpgas. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '13, pages 17–20, Washington, DC, USA, 2013. IEEE Computer Society.

[109] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950.

[110] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 75–84, New York, NY, USA, 2017. Association for Computing Machinery.

[111] Sergej Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. Noftl: Database systems on ftl-less flash storage. *Proc. VLDB Endow.*, 6(12):1278–1281, August 2013.

[112] Tayler H. Hetherington, Mike O'Connor, and Tor M. Aamodt. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 43–57, New York, NY, USA, 2015. Association for Computing Machinery.

[113] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, and Tor M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS '12, pages 88–98, Washington, DC, USA, 2012. IEEE Computer Society.

[114] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[115] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. Pink: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 173–187. USENIX Association, July 2020.

[116] Intel Inc. Intel Data Direct I/O Technology. `http://www.intel.com/content/www/us/en/io/direct-data-i-o.html`.

[117] Intel Inc. Intel Data Plane Development Kit(Intel DPDK) Overview - Packet Processing on Intel Architecture. `http://www.intel.com/content/dam/www/public/us/en/documents/presentation/dpdk-packet-processing-ia-overview-presentation.pdf`, December 2012.

[118] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent distributed storage. *Proc. VLDB Endow.*, 10(11):1202–1213, August 2017.

[119] Zsolt Istvan, Louis Woods, and Gustavo Alonso. Histograms as a side effect of data movement for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1567–1578, New York, NY, USA, 2014. ACM.

[120] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. Yoursql: A high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.*, 9(12):924–935, August 2016.

[121] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 236–243, Washington, DC, USA, 2012. IEEE Computer Society.

[122] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752, Washington, DC, USA, 2011. IEEE Computer Society.

[123] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[124] S. W. Jun, S. Xu, and Arvind. Terabyte sort on fpga-accelerated flash storage. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–24, April 2017.

[125] Sang-Woo Jun, Ming Liu, Kermin Elliott Fleming, and Arvind. Scalable multi-access flash store for big data analytics. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 55–64, New York, NY, USA, 2014. ACM.

[126] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. Bluedbm: An appliance for big data analytics. In *Proceed-*

ings of the 42Nd Annual International Symposium on Computer Architecture, ISCA '15, pages 1–13, New York, NY, USA, 2015. ACM.

[127] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. Bluedbm: Distributed flash storage for big data analytics. *ACM Trans. Comput. Syst.*, 34(3), June 2016.

[128] Sang-Woo Jun, Ming Liu, Shuotao Xu, and Arvind. A transport-layer network for distributed fpga platforms. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–4, Sept 2015.

[129] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 411–424, Piscataway, NJ, USA, 2018. IEEE Press.

[130] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[131] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 437–450, USA, 2016. USENIX Association.

[132] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. The triejax architecture: Accelerating graph operations through relational joins. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1217–1231, New York, NY, USA, 2020. Association for Computing Machinery.

[133] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.

[134] Asif Khan, Muralidaran Vijayaraghavan, Silas Boyd-Wickizer, and Arvind. Fast and cycle-accurate modeling of a multicore processor. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '12, page 178–187, USA, 2012. IEEE Computer Society.

[135] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory ssds. In *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*. Citeseer, 2011.

[136] Myron King, Jamey Hicks, and John Ankcorn. Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA International Symposium on*

Field-Programmable Gate Arrays, FPGA '15, pages 13–22, New York, NY, USA, 2015. ACM.

[137] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash local flash. *SIGARCH Comput. Archit. News*, 45(1):345–359, April 2017.

[138] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A new linux swap system for flash memory storage devices. In *Computational Sciences and Its Applications, 2008. ICCSA '08. International Conference on*, pages 151–156, June 2008.

[139] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 468–479, New York, NY, USA, 2013. ACM.

[140] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 219–231, New York, NY, USA, 2017. ACM.

[141] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, Carlsbad, CA, October 2018. USENIX Association.

[142] Ian Kuon, Russell Tessier, and Jonathan Rose. *FPGA architecture: Survey and challenges*. Now Publishers Inc, 2008.

[143] Maysam Lavasani, Hari Angepat, and Derek Chiou. An fpga-based in-line accelerator for memcached. *IEEE Comput. Archit. Lett.*, 13(2):57–60, July 2014.

[144] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[145] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H. Peter Hofstee, Gi-Joon Nam, Mark R. Nutter, and Damir Jamsek. Extrav: Boosting graph processing near storage with a coherent accelerator. *Proc. VLDB Endow.*, 10(12):1706–1717, August 2017.

[146] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086. ACM, 2008.

[147] Sungjin Lee, Jihong Kim, and Arvind. Refactored design of i/o architecture for flash storage. *Computer Architecture Letters*, 14(1):70–74, Jan 2015.

[148] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 339–353, Santa Clara, CA, February 2016. USENIX Association.

[149] Hans-Otto Leilich, Günther Stiege, and Hans Christoph Zeidler. A search processor for data base management systems. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4*, VLDB '78, page 280–287. VLDB Endowment, 1978.

[150] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[151] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell+: Snapshot isolation without snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 425–441. USENIX Association, November 2020.

[152] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 137–152, New York, NY, USA, 2017. Association for Computing Machinery.

[153] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 476–488, New York, NY, USA, 2015. ACM.

[154] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[155] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association.

[156] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47, New York, NY, USA, 2013. ACM.

[157] Kerstin Lindblad-Toh, Manuel Garber, Or Zuk, Michael F Lin, Brian J Parker, Stefan Washietl, Pouya Kheradpour, Jason Ernst, Gregory Jordan, Evan Mauceli, et al. A high-resolution map of human evolutionary constraint using 29 mammals. *Nature*, 478(7370):476–482, 2011.

[158] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. *SIGPLAN Not.*, 50(4):369–381, March 2015.

[159] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Comput. Surv.*, 52(6), October 2019.

[160] M. Liu, S. Jun, S. Lee, J. Hicks, and Arvind. minflash: A minimalistic clustered flash array. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1255–1260, 2016.

[161] Xin Liu and Kenneth Salem. Hybrid storage management for database systems. *Proc. VLDB Endow.*, 6(8):541–552, June 2013.

[162] Andrea Lottarini, João P. Cerqueira, Thomas J. Repetti, Stephen A. Edwards, Kenneth A. Ross, Mingoo Seok, and Martha A. Kim. Master of none acceleration: A comparison of accelerator architectures for analytical query processing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 762–773, New York, NY, USA, 2019. ACM.

[163] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. Graphssd: Graph semantics aware ssd. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 116–128, New York, NY, USA, 2019. ACM.

[164] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, 2013. USENIX.

[165] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. an analysis of persistent memory use with whisper.

[166] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[167] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford,

Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[168] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[169] John Ousterhout. RAMCloud and the Low-Latency Datacenter. `http://www.snia.org/sites/default/files/JohnOusterhout_RAMCloud.pdf`, 2014.

[170] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[171] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[172] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[173] Jian Ouyang, Wei Qi, Wang Yong, Yichen Tu, Jing Wang, and Bowen Jia. Sda: Software-defined accelerator for general-purpose distributed big data analysis system. In *Hot Chips: A Symposium on High Performance chips, Hotchips*, 2016.

[174] Xiangyong Ouyang, N.S. Islam, R. Rajachandrasekar, J. Jose, Miao Luo, Hao Wang, and D.K. Panda. Ssd-assisted hybrid memory to accelerate memcached over high performance networks. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 470–479, Sept 2012.

[175] M. Owaida, D. Sidler, K. Kara, and G. Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218, April 2017.

[176] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. W. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *2009 IEEE 7th Symposium on Application Specific Processors*, pages 35–42, July 2009.

[177] D. Park, J. Wang, and Y. Kee. In-storage computing for hadoop mapreduce framework: Challenges and possibilities. *IEEE Transactions on Computers*, pages 1–1, 2016.

[178] Ilia Petrov, Guillermo Almeida, Alejandro Buchmann, and Ulrich Gräf. Building large storage based on flash disks. In *Proceeding of ADMS 2010, in conjunction with VLDB 2010*, 2010.

[179] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[180] Mendel Rosenblum and Aravind Narayanan Mario Flajslik. Low latency rpc in ramcloud. `https://forum.stanford.edu/events/2011/2011slides/plenary/2011plenaryRosenblum.pdf`, 2011.

[181] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[182] Zhenyuan Ruan, Tong He, and Jason Cong. Insider: Designing in-storage computing system for emerging high-performance drive. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 379–394, Berkeley, CA, USA, 2019. USENIX Association.

[183] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. Bonsai: High-performance adaptive merge tree sorting. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 282–294. IEEE Press, 2020.

[184] SanDisk. Fusion ioMemory™ PX600 PCIe Application Accelerators. `https://www.sandisk.com/content/dam/sandisk-main/en_us/assets/resources/enterprise/data-sheets/fusion-iomemory-px600-pcie-application-accelerators-datasheet.pdf`. accessed: 2021-01-30.

[185] Mohit Saxena and Michael M Swift. Flashvm: Virtual memory management on flash. In *USENIX Annual Technical Conference*, 2010.

[186] Schuster, Nguyen, Ozkarahan, and Smith. Rap.2—an associative processor for databases and its applications. *IEEE Transactions on Computers*, C-28(6):446–458, June 1979.

[187] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design*

and Implementation, OSDI'14, pages 67–80, Berkeley, CA, USA, 2014. USENIX Association.

[188] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813, 2019.

[189] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct universal access: Making data center resources available to fpga. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, pages 127–140, Berkeley, CA, USA, 2019. USENIX Association.

[190] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 403–415, New York, NY, USA, 2017. ACM.

[191] Malcolm Singh and Ben Leonhardi. Introduction to the ibm netezza warehouse appliance. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '11, pages 385–386, Riverton, NJ, USA, 2011. IBM Corp.

[192] J.R. Spiegel, M.T. McKenna, G.S. Lakshman, and P.G. Nordstrom. Method and system for anticipatory package shipping, December 27 2011. US Patent 8,086,546.

[193] Radu Stoica and Anastasia Ailamaki. Improving flash write performance by using update frequency. *Proc. VLDB Endow.*, 6(9):733–744, July 2013.

[194] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 347–353, Boston, MA, 2012. USENIX.

[195] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. Choosing a cloud dbms: Architectures and tradeoffs. *Proc. VLDB Endow.*, 12(12):2170–2182, August 2019.

[196] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic´. Ramp gold: An fpga-based architecture simulator for multiprocessors. In *Design Automation Conference*, pages 463–468, June 2010.

[197] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 119–132, Berkeley, CA, USA, 2013. USENIX Association.

[198] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally. Darwin-wga: A co-processor provides increased sensitivity in whole genome alignments with high speedup. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 359–372, Feb 2019.

[199] Twitter Inc. Fatcache: memcache on ssd. `https://github.com/twitter/fatcache`.

[200] Michael Ubell. *The Intelligent Database Machine (IDM)*, pages 237–247. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[201] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational query processing on opencl-based fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, Aug 2016.

[202] S. Watanabe, K. Fujimoto, Y. Saeki, Y. Fujikawa, and H. Yoshino. Column-oriented database acceleration using fpgas. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 686–697, April 2019.

[203] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014.

[204] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. The q100 database processing unit. *IEEE Micro*, 35(3):34–46, 2015.

[205] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth 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*, ASPLOS '14, pages 255–268, New York, NY, USA, 2014. ACM.

[206] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN '15, pages 2:1–2:10, New York, NY, USA, 2015. ACM.

[207] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojum Kim, Sungjin Lee, and Arvind. AQUOMAN: An Analytic-Query Offloading Machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 386–399, 2020.

[208] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. Bluecache: A scalable distributed flash-based key-value store. *Proc. VLDB Endow.*, 10(4):301–312, November 2016.

[209] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. A locality-aware energy-efficient accelerator for graph mining applications. MICRO '20, pages 1–13, 2020.

[210] X. Yu, M. Youill, M. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker. Pushdowndb: Accelerating a dbms using s3 computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1802–1805, 2020.

[211] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association.

[212] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association.

[213] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, Boston, MA, June 2010. USENIX Association.

[214] Hanqing Zeng and Viktor Prasanna. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 255–265, New York, NY, USA, 2020. Association for Computing Machinery.

[215] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery.

[216] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.*, 8(11):1226–1237, July 2015.

[217] P. Zhou, Z. Ruan, Z. Fang, M. Shand, D. Roazen, and J. Cong. Doppio: I/o-aware performance analysis, modeling and optimization for in-memory computing framework. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 22–32, April 2018.

[218] Daniel Ziener, Florian Bauer, Andreas Becher, Christopher Dennl, Klaus Meyer-Wegener, Ute Schürfeld, Jürgen Teich, Jörg-Stephan Vogt, and Helmut Weber. Fpga-based dynamically reconfigurable sql query processing. *ACM Trans. Reconfigurable Technol. Syst.*, 9(4):25:1–25:24, August 2016.