Big Data Analytics Made Affordable Using Hardware-Accelerated Flash Storage

by

Sang-Woo Jun

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 2018

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2018

Signature redacted

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

Signature redacted

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

Big Data Analytics Made Affordable Using Hardware-Accelerated Flash Storage

by

Sang-Woo Jun

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Vast amount of data is continuously being collected from sources including social networks, web pages, and sensor networks, and their economic value is dependent on our ability to analyze them in a timely and affordable manner. High performance analytics have traditionally required a machine or a cluster of machines with enough DRAM to accommodate the entire working set, due to their need for random accesses. However, datasets of interest are now regularly exceeding terabytes in size, and the cost of purchasing and operating a cluster with hundreds of machines is becoming a significant overhead. Furthermore, the performance of many random-access-intensive applications plummets even when a fraction of data does not fit in memory. On the other hand, such datasets could be stored easily in the flash-based secondary storage of a rack-scale cluster, or even a single machine for a fraction of capital and operating costs. While flash storage has much better performance compared to hard disks, there are many hurdles to overcome in order to reach the performance of DRAM-based clusters.

This thesis presents a new system architecture as well as operational methods that enable flash-based systems to achieve performance comparable to much costlier DRAM-based clusters for many important applications. We describe a highly customizable architecture called BlueDBM, which includes flash storage devices augmented with in-storage hardware accelerators, networked using a separate storage-area network. Using a prototype BlueDBM cluster with custom-designed accelerated storage devices, as well as novel accelerator designs and storage management algorithms, we have demonstrated high performance at low cost for applications including graph analytics, sorting, and database operations. We believe this approach to handling Big Data analytics is an attractive solution to the cost-performance issue of Big Data analytics.

Thesis Supervisor: Arvind
Title: Johnson Professor, Electrical Engineering and Computer Science

Acknowledgments

I have thoroughly enjoyed my seven years at MIT, surrounded by great, knowledgeable, and overachieving people in an environment that was supportive and well-equipped for research. I will dearly miss it here once I leave.

One of the biggest strokes of luck I've had in regards to my career at MIT was working with my advisor, Professor Arvind. I am sincerely grateful for his exceptionally kind way of interacting with students, and caring personally about all of us. Without his faith and trust in my research projects on accelerated flash storage, which were unfamiliar topics in the group at the time, we would never have been able to build systems and run experiments in the scale in which we have. I would like to thank Arvind for trying his best to equip me with the tools and fundamentals to succeed, and I hope I am capable of retaining enough of them! As I take my first step into academia, I hope someday I will be able to have similar impact on my students. I would also like to thank his wife, Gita, for personally caring about, and caring for, all of us. She was often one of the major reasons morale at the group was very high!

I would also like to thank my thesis committee members, Professor Martin Rinard and Professor Srinivas Devadas, for their effort and insightful feedback. They provided ways of viewing research, as well as writing and presenting it, that was alternative and complementary to Arvind's. I was sure if my research was to be certified by these two professors in addition to Arvind, the ideas and logic could be considered bulletproof. I hope I have grown from their involvement, and managed to reach that goal.

I am thankful for having wonderful colleagues in the group, both past and present. Elliott was there from the beginning to show me the ropes, and Abhinav, Asif, Myron, Nirav, and Richard were seniors I could look up to. Murali started out as such a senior, but quickly became a commiserating and bickering friend through his long tenure at MIT. I thank members of the BlueDBM team and the blood, sweat and tears we have shared: Ming for being the sane voice, Shuotao for outlandish talks about dating, politics and beyond, and Chanwoo for honest opinions about even the

bad things. Thanks to other group members: Andy for diligence and sincerity that puts us all to shame, Sizhuo for expertise and willingness to help, Thomas for all the interesting chats by the whiteboard, and Joonwon for checking in on our office regularly and making sure we are still alive. Also, thanks to Arvind's assistant Sally for keeping the group running.

I've had the fortune of meeting many good friends I hope to keep forever. Thanks to Pablo, Suvinay for commiserating with some daytime drinking. Thanks to Chong-U, Dominic, Law, and Kisuk for all the midnight Towerfall sessions, and for never missing a PAX east! I want to especially thank Tad for all the weekend coffee shop study dates, and to Young-gyu and Sungjun for regularly calling me out to lunch and keeping me from becoming a hermit. Very big gratitude to my hometown friend Yongsuk for coming all the way to Cambridge as an exchange student, so he could matchmake me and Jessica, who ended up becoming my wife. Also to Tushar, Hsin-Jung, Shabnam, Amy, Albert, Po-An, Xiangyao, Yongwook, Jeesoo, Soohyun, Jinyoung and Minjee for sharing various parts of the graduate school career with me.

I would not have had the opportunity to experience life and research at MIT if it was not for the CARES group at SNU, where I interned as a undergraduate research assistant. I am forever grateful for Professor Jihong Kim for starting me on the research path. Thanks to Dr. Sungjin Lee, who was there with me at SNU, and then at MIT, and now a prolific professor back in Korea. Also to Dr. Wook Song and Dr. Jisung Park for all the spent nights at the lab before group meetings taking turns on the folding bed.

I would like to thank Quanta Computers and Samsung, for without their interest, funding, and hardware donations this project would not have been possible. Special thanks to Xilinx for their expertise and continued stream of hardware donations. I also thank the Kwanjeong education foundation for financially supporting my graduate education.

Special gratitude is held for my parents, Yong-Kee Jun and Jeong-Sub Sohn for many many things, among which is their enduring faith in me throughout the long graduate school process, and the uncertainties that existed throughout. I am sincerely

thankful for my lovely wife Jessica, who I was extremely lucky to have met and somehow convinced to marry me during her time as a PhD student at Harvard. Thanks a lot for also weeding out a lot of the grammar errors that used to exist in this document. Let's have a fun and wonderful life together. Last, but absolutely not least, I thank our yet-unnamed son who will be joining the family later this year. I look breathlessly forward to meeting you, and I am glad and relieved I will graduate before I do.

Thank you everyone, from the bottom of my heart.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Complex analytics of large amounts of data, also known as Big Data, has become an integral part of many industries, spanning from business and marketing [31], biomedical and health applications [119], social network analysis [86, 16] to optimal resource management in power grids [54, 192] and terrorist network detection [184]. It is and will continue to be one of the biggest economic drivers for the IT industry.

Big Data by definition does not fit in the main memory (e.g., DRAM) of personal computers or even single server-class machines. Furthermore, meaningful analysis on large amounts of data often requires complex, data intensive queries and algorithms that involve irregular access patterns. One popular approach to handle such queries is a *RAMCloud* [138] style architecture, where a cluster built using fast interconnect has enough collective DRAM to accommodate the entire dataset. A RAMCloud cluster can often deliver high performance for fine-grained random accesses on multi-terabyte datasets. However, the capital and operating cost of such clusters are high, and our experiments agree with common knowledge that performance plummets if even a small fraction of memory accesses spill over into secondary storage.

In this thesis we explore if cheaper and cooler flash storage can provide a viable alternative to RAMCloud, by using architectural modifications including in-storage hardware accelerators. Flash storage is an order of magnitude cheaper per byte compared to DRAM. For example, as of 2018, a 1 TB PCIe-attached SSD device costs around $400, while 1 TB of DRAM costs more than $8,000. Furthermore, such an SSD

consumes less than 10 W of power during active operation, compared to over 200 W consumed by 1 TB of DRAM. If flash storage can be used as a primary datastore for analytics, the cost of high-performance systems can be brought down to a fraction of what it is today. NAND-flash SSDs are better than conventional mechanical hard disk storage for our purpose because SSDs are an order of magnitude faster for sequential accesses and several orders of magnitude faster for random accesses.

There are still many challenges in building flash-based analytics systems. Compared to DRAM, the flash storage fabric achieves relatively low performance, both in sequential and random accesses, and can only be accessed in coarse, kilobyte-range page granularities, compared to the cache-line granularity of DRAM. The coarse-grained granularity issue is especially troublesome, as complex queries with irregular, fine-grained random access may incur a thousandfold performance degradation if bandwidth is wasted. Flash characteristics also include read-write imbalance both in terms of performance and complexity of operations required, and suffer from lifetime issues, where a cell becomes unstable after many write cycles. System architecture-wise, conventional flash-based SSDs are designed as drop-in replacements to legacy hard disks, and require a Flash Translation Layer (FTL) on the storage device to hide flash storage characteristics from the host CPU and expose a disk-like Logical Block Address (LBA) interface. The FTL is a costly component both in terms of hardware resources and operation, and typically requires a multicore ARM processor and multiple GB of DRAM on the storage device dedicated to managing the mapping. Furthermore, many of the storage management libraries and OS components were not designed with such high-performance devices in mind, and often become the bottleneck if not used carefully. All of these issues will be described in detail in Chapter 2.

This thesis demonstrates solutions to many of these issues using a new system architecture as well as algorithms and software systems that achieve high performance analytics in various applications using flash storage as primary memory. Our new system architecture, as well as our prototype implementation of it, is called *BlueDBM* [81, 82]. The key component of BlueDBM is its flash storage device with

in-storage reconfigurable hardware acceleration and rearchitected flash interface. We show that cluster-class performance can be reached by simply plugging a BlueDBM storage device into a PC, and running software that takes advantage of it. BlueDBM can also be efficiently scaled by deploying as a rack-scale cluster, where the hardware accelerators in each storage device are networked together in a dedicated storage area network, separate from the network of the host server machines. Because no off-the-shelf flash storage device provided programmable hardware accelerators and dedicated storage area network capability, we designed and constructed a custom storage device with FPGA-based in-storage hardware accelerators as well as high-performance network interfaces [108]. We have constructed a 20-node prototype cluster with our custom storage device plugged into each node's PCIe slot, and devcloped software systems to help us effectively use the prototype system, including accelerator-aware flash management software, block device drivers and file systems.

Using our prototype cluster, we have explored various applications including high-dimensional nearest-neighbor search, better file systems for flash management, terabyte-scale sorting, key-value stores, and graph analytics systems. In all applications explored on our architecture, analytics problems that previously required datacenter-class clusters can be handled using a rack-scale cluster, or even a single machine, without sacrificing performance. The positive experimental results from a wide array of applications demonstrate that our solutions are viable and attractive for many important applications, suggesting a general trend warranting further research.

It should be noted that the goal of this thesis is to lower the cost of Big Data analytics so that we can achieve analytics of scale that were not affordable with in-memory systems. Situations where in-memory analytics is affordable is not our major focus. For example, when data size is small, or when a large system with enough DRAM can be afforded, conventional approaches can be satisfactory. Our solutions start becoming important when data sizes scale beyond the capabilities of the affordable system.

1.1 BlueDBM Architecture and Prototype Platform

BlueDBM, short for Blue Database Machine, is an alternative system architecture as well as its prototype implementation for high-performance analytics using distributed flash storage. BlueDBM is a rack-scale architecture with distributed flash storage, with three special features: (1) each storage device is augmented with an in-storage hardware accelerator with high-performance low-latency access into the flash storage, (2) the hardware accelerator is able to access the flash chips directly, instead of having to go through the Flash Translation Layer, and (3) the storage devices are networked together in a dedicated storage area network separate from the host server network.

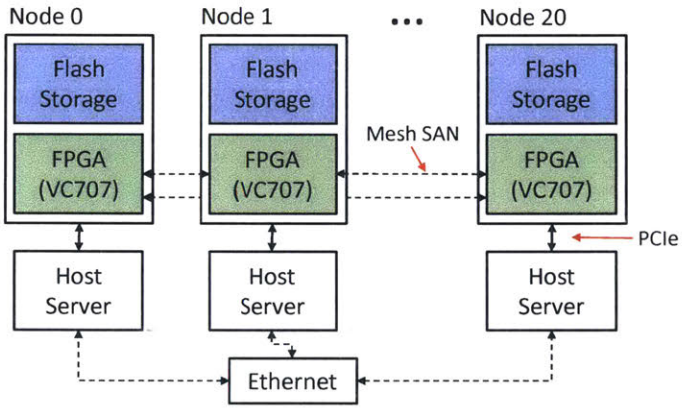


Figure 1-1: BlueDBM prototype cluster architecture

We have constructed a 20-node prototype BlueDBM cluster to evaluate its benefits, using custom-designed PCIe-attached accelerated flash storage devices. The overall architecture of the BlueDBM prototype cluster can be seen in Figure 1-1. Our custom storage device can be seen in Figure 1-2. It is designed to be plugged into a host FPGA board, which acts as the in-storage accelerator, via the FPGA Mezzanine Connector (FMC) interface. We implemented a low-level error-corrected interface to flash chips in the on-board Artix 7 FPGA chip, and it pins out high-throughput multi-gigabit transceivers from the host FPGA in a SATA form factor for the storage area network.

Our prototype cluster of BlueDBM provides the following characteristics:

1. A 20-node cluster with enough flash storage to host Big Data workloads up to

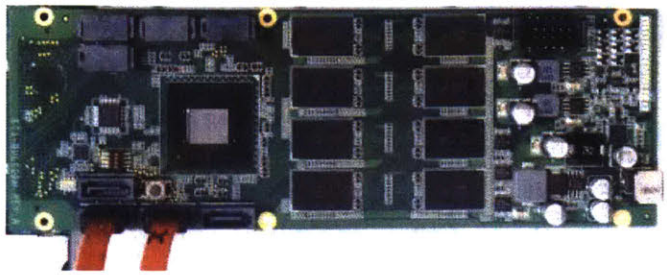


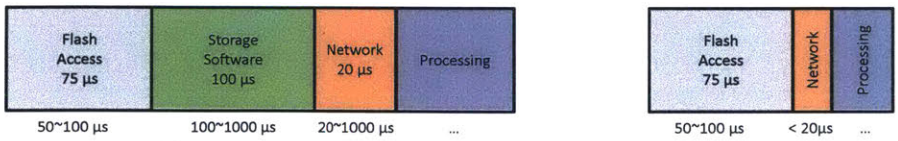
Figure 1-2: Custom-designed minFlash card exposes a custom interface to 0.5 TB of NAND flash storage

20 TBs;

2. Capacity to implement user-defined in-store processing engines using hardware acceleration;
3. Flash storage hardware and firmware designs which expose an interface to implement application-specific optimizations in flash access;
4. High-performance storage-area network to achieve near-uniform latency access into a network of storage devices that form a global address space.

The characteristics of BlueDBM improve system performance in the following ways:

Latency: BlueDBM achieves extremely low latency access into distributed flash storage. Network software stack overhead is removed by integrating the storage area network into the storage device, and network latency is reduced using a faster network fabric. It reduces the storage access software latency with cross-layer optimizations or removes it completely via in-storage processing engine. It reduces processing time using application-specific accelerators. The latency improvements are depicted in Figure 1-3.



(a) Generic System (b) BlueDBM System

Figure 1-3: Latency breakdown of distributed flash access

Bandwidth: Bandwidth of storage access is increased. In conventional architectures, it is often challenging to write software that makes efficient use of the multi-gigabytes of available flash bandwidth, both due to the difficulty of writing software that can consume that much bandwidth, and inefficiencies of existing interface into flash storage for access patterns that are not large, contiguous accesses. BlueDBM gives application the option to optimize internal flash management to fit the access patterns of the application, and also exposes a high-performance asynchronous interface that achieves high performance even with small, random accesses. An application-specific accelerator in the storage can also consume data read at device speed, instead of being bound by the software performance or the performance of the PCIe link to the host.

Power: Power consumption of the overall system is reduced. An in-store processor sometimes removes the need for data to be moved to host, reducing power consumption related to data movement. Flash storage consumes far less power compared to DRAM of comparable capacity. Application-specific accelerators are also more power efficient than a general purpose CPU or GPU.

Cost: Cost of the overall system is reduced. The engineering cost of constructing a separate accelerator is reduced by integrating the accelerator into the storage device. The cost of flash storage is also much lower than DRAM of comparable capacity.

As we will discuss in Chapter 2, almost every element of our system is present in some commercial system. Yet our system architecture as a whole is unique. The various details about the BlueDBM architecture and its prototype are described in Chapter 3 to Chapter 5.

1.2 Application Evaluation on BlueDBM

We have evaluated analytics on flash storage using the BlueDBM architecture by implementing and evaluating various applications and system software on the BlueDBM prototype. In all of the application evaluations we have explored, we were able to achieve performance comparable to much costlier DRAM-based clusters, and some-

times even exceeded them in some situations, using much fewer and cheaper BlueDBM nodes. The applications span from relatively straightforward external sorting to more challenging applications including a terabyte-scale graph analytics system called GraFBoost. This suite of applications highlight the capacity of a hardware-accelerated flash-based system in analytics as well as data management to achieve high performance at a relatively low cost. Of the six applications described below, I am the primary designer of four which will be introduced in detail from Chapter 6 to Chapter 9. The other two applications have been led by others, and also reinforce the appeal of BlueDBM.

1.2.1 High-Dimensional Nearest-Neighbor Search

We developed approximate nearest-neighbor search using Locality Sensitive Hashing [58]. Data is stored in flash storage, and the distance metrics are implemented in the in-storage hardware accelerator. We have evaluated three distance metrics of varying complexity: bitwise hamming distance, cosine similarity, and image comparison using joint color histograms.

For a relatively complex image comparison distance metric, our implementation outperformed a disk-based system by 10x, and even a DRAM-based multicore system by 1.3x. For a simple hamming distance metric, the in-memory software system performed well until not all of the data was able to fit in memory. Even when 10% of accesses spilled over into storage, performance dropped sharply to the point it was comparable to our flash-based implementation. Meanwhile, our system consumes about half of power per node compared to a DRAM-based system, and less than half in terms of power consumption per amount of processed data.

In terms of data access, this application is coarse-grained random access intensive and not sensitive to latency, and has moderate to high computational overhead according to the distance metric.

1.2.2 Terabyte-Scale External Sort

We developed external merge-sort using a sorting network-based merge-sort accelerator. Our merge-sort accelerator is composed of multiple merge-sort accelerators optimized to the four types of memory fabric in the storage hierarchy: on-chip registers, on-chip block RAM, off-chip DRAM, and flash storage. Each accelerator uses multi-rate merge-sorters based on sorting networks to saturate the bandwidth of the memory fabric it operates on. As a result, our implementation achieves very high single-node performance, in comparison to the published numbers of a 21-node Hadoop cluster, at a fraction of power consumption. In terms of performance per power consumption, even our unoptimized prototype implementation was able to rival the power-performance of the current JouleSort [149] champion. A system with a power-efficient embedded processor and two flash cards is projected to achieve over twice the power-performance of the JouleSort champion.

This application is bound by the bandwidth of sequential access, as well as computation overhead of merge-sorting.

1.2.3 Wire-Speed Aggregator for Database Acceleration

Aggregation operations are one of the primary operations for relational databases, used by GROUPBY queries to collapse multiple rows and extract higher level information by applying the user-specified aggregation functions (like ADD, MAX, or AVERAGE) between values that share the same row key. Our new aggregator design overcomes two limitations of existing designs for hardware aggregators: it achieves wire-speed even when (1) the aggregation function has multi-cycle latency, and (2) the wire is wide, meaning multiple elements arrive every cycle. As a result, a single instance of the new aggregator design is fast enough to saturate the PCIe storage device or the network backing the database. While this application does not explicitly involve flash storage, it is an integral component of the graph analytics system introduced below, and will be one of the key components of a relational database accelerator we will develop in the future.

1.2.4 GraFBoost Graph Analytics System

One of the most prominent applications we have explored on the BlueDBM architecture is terabyte-scale graph analytics. Due to the fine-grained random access nature of graph analytics algorithms, previous competitive systems for solving large-scale graph analytics problems have taken one of two approaches: (1) provision the system with enough DRAM to store the entire graph data structure, then perform random accesses to the graph directly out of DRAM, or (2) provision the system with enough DRAM to store only the vertex data in DRAM, then stream the edges in from secondary storage. As graphs of interest reach multiple terabytes, and are expected to grow further, systems with enough DRAM for these graphs is becoming ever more expensive. Furthermore, our experiments show that their performance drops sharply when the graph size exceeds available memory capacity.

As an alternative to such a costly system, we present GraFBoost, a flash storage based system for large-scale graph analytics computations that achieves high performance while storing and accessing both vertex data and graph structure in flash storage. GraFBoost implements (in both software and as a hardware accelerator) a novel algorithm called sort-reduce, which sequentializes fine-grained random updates.

We compare the performance of the two single-node GraFBoost implementations with single-node and small cluster implementations of other modern graph analytics systems, using algorithms implemented by the same developers. We have tested all systems using a suite of synthetic and real-world graphs, ranging from 6 GB to 502 GB, and show that GraFBoost (both software only and hardware accelerator versions) was the only system that could process all of our benchmarks successfully on all graphs. No other system performed all of our benchmark graph computations successfully within a reasonable time budget of 30 minutes. Even for smaller graphs for which other systems could complete successfully, GraFBoost demonstrated performance comparable to the fastest systems evaluated. These results highlight the ability of GraFBoost to scale to large graph computations independently of the amount of DRAM and with predictable performance.

In terms of data access, this application is bound by fine-grained random read-modify-writes, but the sort-reduce algorithm turns it into a problem bound by sequential bandwidth and computation. The computation overhead is handled by the in-storage accelerators.

1.2.5 Refactored I/O Architecture and File System

Because I was not the primary researcher driving this project, its details are not covered in this thesis. Detailed information about the refactored I/O architecture and file system can be found in the primary papers by Sungjin Lee et al. [98, 100]

The *RE*factored *D*esign of I/*O* architecture (REDO) [98] and its BlueDBM flavor Application-Managed Flash (AMF) [100] are novel approaches to managing flash storage, by moving the intelligence of flash management from devices to applications, which can be file systems, databases and user applications. Only the essential flash management functionality is left on the flash storage device, such as providing an error-free interface, bad block management and wear-leveling. On the BlueDBM platform, AMF implements a new block I/O interface which does not support overwrites of data unless they were explicitly deallocated. This dramatically simplifies the management burden inside the device because fine-grained remapping and garbage collection do not have to be done in the device. Also, applications can use high-level information available to make intelligent decisions in flash management, including efficient scheduling for regular cleaning I/Os, and accurately separating hot and cold data.

We have evaluated AMF on BlueDBM using a new file system implementation called Application-managed Log-structured File-System (ALFS), which is very similar to conventional Log-structured File Systems (LFS), except it appends the metadata as opposed to updating it in-place. Our experiments of running database benchmarks on top of ALFS show that it improves I/O performance by 80%, and improves storage lifetime by 38%, over conventional FTLs. The DRAM requirement for the FTL was reduced by a factor of 128 because of the new interface while additional host-side resources (DRAM and CPU cycles) required by AMF were minor.

1.2.6 BlueCache: Flash-Based Key-Value Cache

Because I was not the primary researcher driving the BlueCache project, the details of BlueCache is not covered in this thesis. Detailed information about BlueCache and its evaluation can be found in the primary paper by Shuotao Xu et al. [185]

A key-value store (KVS), such as memcached and Redis, is widely used as a caching layer to augment the slower persistent backend storage in data centers. 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. DRAM-based KVS provides fast key-value access, but its scalability is limited by the cost, power and space needed by the machine cluster to support a large amount of DRAM.

The high density, low cost per GB and low power consumption of flash storage make flash-based KVS a viable, cost-effective alternative for scaling up the KVS cache size. However, flash storage has orders of magnitude higher latency compared to DRAM, in the range of 100μs compared to 10-20ns. 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 [132]. Netflix's EVCache KVS cluster has 99th percentile latency of 20 ms and is still able to deliver rich experience for its end users [130].

BlueCache is 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; and (4) Elimination of FTL with a log-structured KVS flash manager, which implements simple garbage collection and schedules out-of-order flash requests to maximize parallelism. The architectural

exploration required to develop BlueCache was possible by BlueDBM.

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 [169].
- 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.

This application is bound by the performance of fine-grained and also coarse-grained read and writes, as most of computation and network overheads have been removed by hardware accelerators and the low-latency network.

1.3 Thesis Contributions and Organization

This thesis suggests the viability of using cheaper flash storage instead of costly DRAM in Big Data analytics, by evaluating multiple important applications on a novel architecture based on flash storage and in-storage hardware acceleration. This thesis can be divided into two parts.

1.3.1 Part I: A Platform for Flash-Based Analytics

The first part involves the design and implementation of the BlueDBM architecture for high-performance Big Data analytics using flash storage. The following list summarizes the contributions of Part I:

1. Design and implementation of a scalable flash-based system with a global address space, in-store computing capability and a flexible inter-controller network.
2. A hardware-software codesign environment including storage management for incorporating user-defined in-store processing engines.

Part I is organized as follows:

- Chapter 2 introduces background information and existing research on analytics using flash storage, as well as reconfigurable hardware accelerators. Background and related works about applications explored in Part II will be given in their respective chapters.
- Chapter 3 describes the detailed design of BlueDBM, our platform for exploring distributed flash-based system architectures for Big Data analytics. The BlueDBM architecture includes a flash storage device with in-storage acceleration, as well as a file system that is aware of flash storage and reconfigurable hardware accelerators.
- Chapter 4 describes BlueDBM's dedicated storage-area network between in-storage hardware accelerators. The BlueDBM network implements transport-layer protocol semantics in hardware, to provide high-level FIFO-like semantics for multiple virtual channels to the hardware accelerators. By implementing a separate storage area network in the in-storage accelerator, the accelerators can access the entire storage space at sub-microsecond latency.
- Chapter 5 introduces our prototype implementation of BlueDBM, and the results of various performance benchmarks demonstrating the efficiency of the architecture and implementation.

1.3.2 Part II: Application Evaluations

The second part involves demonstrating the viability and attractiveness of analytics on flash storage, by evaluating various applications on BlueDBM. The following list summarizes the contributions of Part II:

1. Development and evaluation of two applications that perform coarse-grained accesses (nearest-neighbor search and sort) and demonstrate the viability of a flash-based system in terms of both performance and cost.
2. A novel sort-reduce algorithm for efficiently sequentializing fine-grained random updates.

3. Development and evaluation of a graph analytics system storing all data in flash storage, by using either a software or hardware implementation of sort-reduce to sequentialize vertex updates, demonstrating the viability of flash-based systems for applications with fine-grained updates.

The organization of Part II is as follows:

- Chapter 6 describes the design, implementation and evaluation of the approximate high-dimensional nearest-neighbor search application using locality sensitive hashing.
- Chapter 7 describes the design, implementation and evaluation of the system for our hardware-accelerated external sorting system. Topics covered include the design of multi-rate merge-sort accelerators for different memory fabrics. This application also serves as an important component of the GraFBoost graph analytics system in Chapter 9.
- Chapter 8 describes the design, implementation and evaluation of the database aggregation accelerator, which uses a novel design based on sorting networks to perform GROUPBY database operations at wire-speed, even when the aggregation function has multiple cycles of latency, and when multiple rows are ingested every cycle. This application serves as an important component of the GraFBoost graph analytics system in Chapter 9.
- Chapter 9 describes GraFBoost, a graph analytics system that uses flash storage and the novel sort-reduce algorithm implemented either in software or as a hardware accelerator, to achieve high performance with very little memory and processing resources.
- We conclude and present possible future research directions in Chapter 10.

Chapter 2

Background

This chapter provides background information and introduces existing research related to flash storage and reconfigurable hardware accelerators.

2.1 Flash Storage

In Big Data scale workloads, one of the most important factors of system performance is fast access to data. Because access to secondary storage is slower than DRAM-based system memory, building a cluster with enough DRAM capacity to accommodate the entire dataset can be very desirable but expensive. An example of such a system is RAMCloud, which is a DRAM-based storage for large-scale datacenter applications [138, 152]. RAMCloud provides more than 64 TB of DRAM storage distributed across over 1000 servers networked over high-speed interconnect. Although RAMCloud provides 100 to 1000 times better performance than disk-based systems of similar scale, its high energy consumption and high price per GB limit its widespread use except for extremely performance-and latency-sensitive workloads.

NAND-flash-based SSD devices are gaining traction as a viable technology to counter the high cost of DRAM-based systems. Flash storage is one of the fastest evolving device technologies today. SSDs are an order of magnitude cheaper both in price per capacity and power consumption compared to DRAM, while achieving an order of magnitude faster performance compared to disk [30]. The performance

of modern flash storage devices have exceeded the capabilities of legacy storage interfaces such as SATA, and there is active research in exploring faster interfaces. PCIe-attached SSD devices have now become common, thanks to industry standards like NVMe [64]. Attempts to use flash as a persistent DRAM alternative by plugging it into DIMM slots are also being explored [123]. Many existing database and analytics software have shown improved performance with SSDs [42, 83, 99]. While the slow performance and long seek latency of disks prohibited serious use of external algorithms, the higher performance of flash storage is putting high performance external analytics within reach. However, there are still challenges to overcome in order for high-performance flash-based analytics, including coarse granularity of access, read-write imbalance in performance and complexity, and limited write lifetime.

2.1.1 Flash Storage Characteristics

Flash-based SSD storage devices have been largely developed as a faster drop-in replacement for hard disk drives, which required special hardware and software to hide NAND-flash characteristics. The two most prominent characteristics are read-write imbalance and lifetime issues. NAND-flash can be read and written in *page* units, which is typically 4 KB to 8 KB in size [30]. While page reads can be done without much restrictions, page writes can only be done to pages that are *erased*, an expensive operation of even coarser *block* granularity of megabytes. Random updates handled naively will cause a block erase every time, requiring all non-updated data to be read and re-written to the erased block. Furthermore, flash also suffers from lifetime issues due to erase and write causing physical wear in its storage cells. As flash cells become older due to repeated write and erase cycles, the bit error rate (BER) of each page steadily increases, until it becomes unusable.

Attempts to hide flash storage characteristics exist in many layers of the system. At the storage device level, commercial flash storage devices handle write issues, wearing and many more issues using an intermediate software called the Flash Translation Layer (FTL) [40]. The FTL manages a mapping from a logical address space to physical pages on flash chips, and exposes a familiar Logical Block Address (LBA)

interface [10]. In the FTL, page updates can be handled by writing to any available erased page, and changing the mapping. When the mapping changes, pages with old values are marked invalid.

When the number of invalid pages becomes large, the FTL also performs *garbage collection*, where it chooses blocks with only a few valid pages in it, copies valid pages inside the blocks to a new location, and erases the whole block for future use. This may cause *write amplification*, where a certain number of page write requests results in many more write operations, negatively impacting performance.

The FTL also performs *wear leveling*, where it tries to assign writes to lesser-used cells. When a page has bit errors, the FTL uses error correcting code (ECC) such as Reed-Solomon or Bose-Chaudhuri-Hocquenghem codes to correct them. Each page in a flash chip is coupled with a few bytes of additional space by default, where the FTL can store checksums created by the ECC implementation. When uncorrectable levels of bit errors occur in a page, the FTL must mark the page as bad and avoid its use.

Flash storage devices achieve high performance by organizing many flash chips into multiple buses and exploiting the parallelism between buses. Another source of parallelism is across chips on the same bus, in order to overlap the latency of access between a flash command and actual data movement. The FTL also tries its best to take advantage of intra-device parallelism, by organizing the mapping so that consecutive accesses span across different buses or chips.

Above the storage device layer, another area of interest is flash-optimized file systems such as log-structured file systems [150, 96]. Many of these efforts have had significant impact in improving flash usability, performance and lifetime. However, some operations like random updates are often still expensive [30].

Due to the size of the map and the computation involved, an FTL involves significant amounts of hardware and software, including multi-GB of DRAM as well as 8 or more cores of ARM processors on the storage device. Going through this additional translation layer adds considerable latency in access. Furthermore, there is functional duplication between FTL and flash-optimized file systems, as each entity

try to do what it thinks is best for performance and lifetime, with the information available to it [97]. These separate efforts sometimes work against each other, leading to inefficient management. Especially for issues such as exploiting parallelism and garbage collection, the high-level knowledge available at the file system is useful for a more efficient logical-physical mapping [97, 100]. However, the efforts of the file system may be counteracted by the FTL, which acts on its own limited information. To overcome these issues, there are active studies in file systems that manage flash chips directly in order to remove the overhead of FTLs [136, 68, 100]. These methods of letting the file system, or even the database application make decisions based on high-level information have been shown to be effective.

2.1.2 Issues in Flash-Based Analytics

Despite improving performance and management, characteristics of flash storage still pose serious challenges in developing external algorithms, including a much lower bandwidth compared to DRAM, higher latency in the range of hundreds of microseconds compared to tens of nanoseconds of DRAM, and a coarser access granularity of 4KB to 8KB *pages*, compared to DRAM's cache line [30]. This difference of efficiency between flash and DRAM can be seen in Figure 2-1. Large access granularity can create a serious storage bandwidth problem for fine-grained random access; if only 4 bytes of data from an 8KB flash page is used, the bandwidth is reduced effectively by a factor of 2048. As a result, naïve use of secondary storage as a memory extension will result in a sharp performance drop, to the point where the system is no longer viable.

Due to the high performance of SSDs, even inefficiencies in the storage management software become significant, and optimizing such software has been under active investigation. Moneta [28] modifies the operating system's storage management components to reduce software overhead when accessing NVM storage devices. Willow [156] provides an easy way to augment SSD controllers with additional interface semantics that make better use of SSD characteristics, in addition to a backwards compatible storage interface.

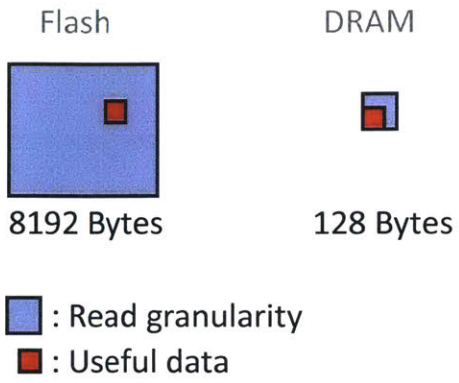


Figure 2-1: Fine-grained access into page-granularity flash results in a lot of wasted bandwidth

The high performance of SSDs also affect the network requirements. The latency to access disk over Ethernet was dominated by the disk seek latency. However, in a SSD-based cluster, the storage access latency could even be lower than network access. These concerns are being addressed by faster network fabrics such as 10GbE and Infiniband [14], and by low-overhead software protocols such as RDMA [107, 75, 148, 175, 107, 147] or user-level TCP stacks that bypass the operating system [79, 69]. QuickSAN [29] is an attempt to remove a layer of software overhead by augmenting the storage device with a low-latency NIC, so that remote storage access does not need to go through a separate network software stack.

2.1.3 In-Storage Processing

Another important attempt to accelerate SSD storage performance is in-storage processing, where some data analytics is offloaded to embedded processors inside SSDs. These processors have extremely low-latency access to storage, and help overcome the limitations of the storage interface bus. The idea of in-storage processing itself is not new. Intelligent disks (IDISK) connected to each other using serial networks have been proposed in 1998 [85], and adding processor to disk heads to do simple filters has been suggested as early as in the 1970s [101, 141, 17]. However, performance improvements of such special-purpose hardware did not justify their cost at the time.

As data sizes become large and more resident in secondary storage, there has been

a lot of work in near-storage processing, where computation is moved to the storage to reduce data movement overhead [76, 65, 36, 93, 48, 49, 84, 156]. These devices have shown promising results, but their gains are often limited by the performance of the embedded processors in such power-constrained devices. One solution under examination is embedding reconfigurable hardware in storage devices [145, 81]. For example, Ibex [178] is a MySQL accelerator platform where a SATA SSD is coupled with an FPGA. Relational operators such as selection and group-by are performed on the FPGA whenever possible; otherwise they are forwarded to software. Companies such as IBM/Netezza [160] offload operations such as filtering to a reconfigurable fabric near storage. On the other end of the spectrum, systems such as XSD [35] embed a GPUs into SSD controllers, and demonstrate high performance accelerating MapReduce. Similar efforts are being made for DRAM as well [137, 39].

2.2 Reconfigurable Hardware Acceleration

Performance scaling of CPUs is slowing down due to power density limitations of silicon-based computation fabric [51], and reconfigurable hardware accelerators are being viewed as one of the most prominent technologies for continued performance scaling. The most prominent reconfigurable hardware accelerator technology is Field-Programmable Gate Arrays (FPGA). FPGA chips contain millions of programmable logic blocks, which can be configured in the field to collectively act as application-specific circuits. Modern FPGAs are also equipped with a large amount of on-chip memory (block RAM, or BRAM) and circuits dedicated to often-used functionality, such as PCIe, Ethernet and other memory interfaces. Modern FPGAs also usually include a large number of Digital Signal Processing (DSP) blocks, which are dedicated circuits used to implement fast signal processing or floating point operations.

While FPGAs generally run at a much lower clock frequency compared to server-class CPUs, creating dedicated hardware allows the circuit to exploit much more parallelism in the application, resulting in over an order of magnitude performance improvements in many important applications. Dedicated circuits on FPGAs also

consume an order of magnitude less power compared to software running on CPUs. Circuits for FPGAs can be compiled in minutes to hours, and the chip can be programmed in milliseconds, which is desirable compared to the long, expensive design cycle of Application-Specific Integrated Circuits (ASIC). Recent advances in partial reconfiguration in FPGAs allow parts of the circuit to be reconfigured to perform different functions while the rest of the chip is active. There is active research in many application domains to use reconfigurable hardware acceleration to improve performance and reduce power consumption.

2.2.1 Application Acceleration Using FPGAs

Database acceleration using reconfigurable hardware accelerators is an actively researched topic. There has been a great amount of interest in FPGA-based query acceleration engines, which handle a subset or all of the four major database queries: projection, restriction, aggregation and join, as well as some data management functions like row decompression [162]. Domain-specific processors for database queries are prominent topics for this purpose [163, 176]. Such projects includie Q100 [180], which is a data-flow style processor with an instruction set architecture that supports SQL queries, and LINQits [38], which maps a query language called LINQ to a set of accelerated hardware templates on a heterogeneous SoC (FPGA + ARM). Both designs exhibited an order of magnitude performance gains at lower power, affirming that specialized hardware for data processing is very advantageous. Glacier [126] has explored taking a query and compiling it into a dedicated accelerator. Some have focused on using dynamic partial reconfiguration functionalities to adapt to the query being accelerated [47, 46, 21]. Systems like Ibex and others have explored database accelerators on the storage device [178, 90]. IBM's Netezza [160] is a commercial product that allows query offloading to a near-storage accelerator. Other systems have implemented accelerators on FPGA fabric which shares main memory with the CPU [139, 159].

The interest in reconfigurable hardware acceleration is not limited to databases, and the benefits of FPGA acceleration are being explored across all application do-

mains. For example, FPGA-based accelerators are being deployed in the finance area both for computational acceleration [44, 179, 72], and for reducing the latency of High-Frequency Trading (HFT) actions. Bioinformatics is another computation-intensive field where hardware acceleration is actively being pursued. Accelerators have been built for various applications including K-means algorithms for microarray data [70], string matching [43] and complex distance algorithms like Smith-Waterman [103]. FPGA-accelerated clusters have been used to accelerate large-scale problems like DNA sequencing and alignment [32, 77]. FPGAs are becoming prevalent in high-throughput scientific applications as well, where the rate of data flow is so fast that general-purpose CPUs cannot keep up. Some examples include radio and optical astronomy [143, 142], and high-throughput particle physics at the European Organization for Nuclear Research (CERN) [13, 63, 24].

2.2.2 Distributed FPGA Clusters

FPGAs offer very desirable performance and power characteristics, but modern data-intensive applications often require more resources than are available on a single FPGA chip. As a result, exploration of distributed FPGA computing systems is gaining popularity. The scale of distributed FPGA deployments range from a cluster-in-a-box systems such as BlueHive [125], to rack-level deployments such as Maxwell [19], and to datacenter-scale deployments such as Catapult [146]. Some have also attempted heterogeneous deployments including GPUs and FPGAs [167], or to insert FPGA accelerators into the storage datapath [81, 82]. Such systems offer a much better power performance characteristics over their off-the-shelf server counterparts.

The TCP/IP network protocol stack is by far the most popular protocol for internetworking computer systems, but it may not be a good fit for inter-FPGA communication as it is a complex and resource-heavy protocol designed for an unpredictable network such as the Internet. Some FPGA cluster projects have used Ethernet's physical and data link layers for their networks, but full implementation of the TCP/IP stack is rare unless it has to interact with a legacy interface [23]. Datacenter-scale protocols such as Infiniband [14] implement a more efficient transport layer protocol

on top of a more reliable network layer implementation. It also offloads major parts of the protocol to the NIC to achieve higher performance. Some have modified the TCP protocol [12] to significantly reduce the packet buffer size using intelligent congestion control.

BlueLink [165] demonstrated that a new protocol using high-speed serial links has a better area-performance characteristics than implementing existing network protocols. Many distributed FPGA computing systems have demonstrated high performance with FPGA nodes networked over such high-speed serial links [26, 19]. Some have developed meta language compilers that generate application-specific network logic, including features like flow control, from separate network specifications [57].

2.2.3 FPGA Acceleration in the Datacenter and Cloud

There is immense interest in industry in incorporating reconfigurable hardware into datacenters and cloud infrastructures. Microsoft is deploying all new Azure and Bing servers with FPGAs installed [146, 34]. Catapult installs FPGAs into the network interface cards (NIC) of each node, so that accelerators have fast and low-latency access to data. Intel, ever since acquiring Altera, is focusing on datacenter architectures with both Xeon processors and FPGAs plugged into the motherboard [3]. For example, Intel is working with Alibaba to install FPGAs in the Alibaba cloud [4]. Such approaches have drastically reduced the cost and power consumption of datacenters while improving performance. The Amazon EC2 cloud service recently started providing instances equipped with FPGAs [1]. Amazon provides build tools, as well as a library of base functionality for accelerator development, including the PCIe link, and the interfaces into network and memory. The availability of FPGA-equipped cloud instances and their popularity in the datacenter dramatically increase the availability of FPGA resources to developers and researchers, and immensely bring down startup cost of hardware acceleration.

Part I

A Platform for Flash-Based Analytics

Chapter 3

BlueDBM: Distributed Flash Store for Big Data Analytics

BlueDBM is a system architecture, as well as a prototype platform for distributed analytics using flash storage and in-storage hardware acceleration. BlueDBM has enabled all research presented in this thesis, such as the GraFBoost graph analytics platform. It is designed to allow unprecedented amount of flexibility in exploring flash storage architecture. Using BlueDBM, we were able to design various accelerated flash-based systems for many important applications, and demonstrate its cost and performance benefits against costlier DRAM-based systems using an actual hardware implementation. The three key requirements in the design of BlueDBM are as follows:

1. Customizable flash management: The flash management functionality in the flash storage device must be open to modification, including the Flash Translation Layer (FTL).
2. In-storage hardware acceleration: A platform for hardware accelerator should exist in-storage.
3. Dedicated storage-area network: The storage devices should be able to communicate with each other directly, without having to go through the host.

It was not possible to achieve these requirements using off-the-shelf components. Internals of FTLs are closely guarded industry secrets, and no commercial storage

device allowed us to change them. At the time of BlueDBM's design, there was also no available platform for in-storage hardware acceleration research, as well as for sideband storage-area networks. As a result, we designed and implemented a custom flash storage device.

We have worked with Quanta Computers Inc., as well as Xilinx Inc. to design and fabricate a flash storage card that meets the above requirements. This chapter describes the design of the BlueDBM architecture, implementation of a prototype 20-node cluster with 20 TB of flash storage, as well as its measured performance with various workloads.

This chapter is organized as follows: Section 3.1 describes the design of the BlueDBM architecture, focusing on the four services provided for the in-storage processor by the BlueDBM platform — the flash access interface, network interface, host interface, and DRAM interface. Section 3.2 describes the software and file system interface of BlueDBM, and how a hardware-accelerated software application can use the file system with the in-storage processor to easily apply acceleration to files. Section 3.3 provides a summary of the chapter and concludes.

3.1 System Architecture

This section describes the design of the BlueDBM architecture and its internal functional modules. How these functions map to actual hardware in our prototype is described in Chapter 5.

The BlueDBM architecture is a homogeneous cluster of host servers coupled with a BlueDBM storage device, as described in Figure 1-1. Each BlueDBM storage device is plugged into the host server via a PCIe link, and it consists of flash storage, an in-storage processing engine, multiple high-speed network interfaces and on-board DRAM. The host servers are networked together using Ethernet or other general-purpose networking fabric. The storage devices are networked together via a dedicated storage area network directly between the in-storage processors. The host server can access the BlueDBM storage device via a host interface implemented over PCIe. It can

either directly communicate with the flash interface to treat it as a raw storage device, or with the in-storage processor to perform computation on the data. Furthermore, it can access remote storage devices as if accessing a local storage device, by issuing a remote access request to the local storage device, which relays the command and data over the storage-area network.

The core of the BlueDBM storage device is the in-storage processor, which can be used to implement anything from a conventional FTL to application-specific hardware accelerators that use the flash storage and network to perform complex computation on terabytes of data. The in-storage processing engine has access to four major services provided by the BlueDBM platform: The flash interface, network interface, host interface and the on-storage DRAM buffer. The host server, PC or laptop can access the in-storage processor through the host interface, and the in-storage processor can access the system components like flash, network and DRAM memory via their respective interfaces. Figure 3-1 shows the four services available to the in-storage processor. In the following sections, we describe the flash interface, network interface and host interface. We omit the DRAM buffer because there is nothing special about its design.

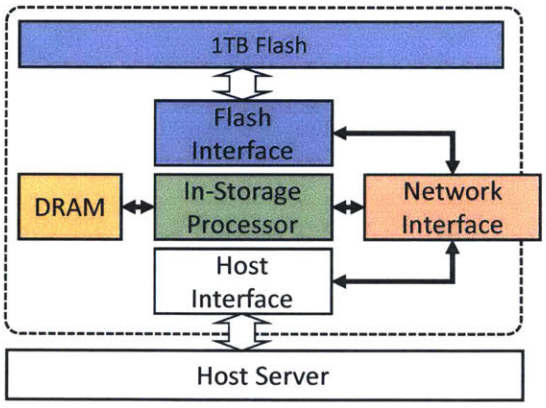


Figure 3-1: BlueDBM node architecture

3.1.1 Custom Flash Storage and Interface

One of the major differences of the BlueDBM storage device compared to conventional off-the-shelf SSDs is the ability to customize, or even entirely remove the FTL. This flexibility allows research of drastically different FTL designs in order to explore more effective methods of managing flash storage.

While the approach of hiding flash characteristics and exposing a backwards-compatible block device interface using the FTL has been beneficial for the widespread adoption of flash storage, it has been discovered to exhibit inefficiencies as well. First of all, this additional layer of management requires a significant amount of hardware and software to operate [97]. Also, FTL also adds considerable latency to access. Furthermore, an FTL tries its best to optimize for performance and lifetime with information available to it, but its optimizations are still inefficient as it operate only using low-level information available at the device level.

As a response to these issues, we chose to move flash management away from the device and into the file system or the block device driver. This approach is discussed in Section 3.2. However, it is worth noting that it is absolutely possible to implement a traditional FTL in BlueDBM, if the aim is to make it act like a conventional off-the-shelf storage device.

In order to have the freedom to modify the FTL, we have built our own custom storage device with such provisions, which we call **minFlash** [108]. The details of minFlash is described in the following sections.

3.1.1.1 Interface for High Performance Flash Access

Our flash controller exposes a low-level, thin, fast and bit-error-corrected hardware interface to raw NAND flash chips, buses, blocks and pages. The flash controller uses physical flash addresses to expose organization of the entire distributed flash array. This has the benefit of (1) cutting down on access latency from the network and in-storage processors; (2) exposing all degrees of parallelism (across buses and chips) of the device and (3) allowing higher level system stacks (file system, database

storage engine) to perform flash managment, using higher-level information available to them.

To access the flash storage, the in-storage processor issues a flash command to the controller with the operation, the physical address and a tag to identify the request. The physical address of a flash page is indicated using the bus number, chip number in the bus, block number in the chip, and finally the page index in the block. The results of the command are returned with the tag of the issued command. The flash controller exposes the commands in Table 3.1.

Command	Description
ReadPage(tag, bus, chip, block, page)	Read a page. Data will be streamed out via **ReadData**.
WritePage(tag, bus, chip, block, page)	Writes a page Pages must be erased before being written, and page writes within a block must be sequential. After a page write request is sent, the user must wait until a *ready to write* acknowledgement is received via **Ack** before streaming in data via **WriteData**.
EraseBlock(tag, bus, chip, block)	Erases a block.
Ack(tag)	Receives an acknowledgement for the completion of a certain tag. The status may indicate correct completion, ready to write, bad blocks on erase, or uncorrectable error on reads.
ReadData	Returns a **[tag,data]** tuple for data read from a page.
WriteData(tag,data)	Write to page indicated by **tag**.

Table 3.1: Flash controller interface

For maximum performance while reading, the controller may send these data bursts *out of order* with respect to the issued request *interleaved* with other read requests. Thus completion buffers may be required somewhere in the datapath, either in the in-storage processor or the host machine, to maintain FIFO characteristics. Furthermore, we note that to saturate the bandwidth of the flash device, multiple

commands must be in-flight at the same time, since flash operations can have latencies of 50 μs or more.

A flash chip handles write operations in multiple phases to hide latency, and minFlash exposes this interface directly. After issuing a flash write request, data cannot be streamed in immediately. Instead, the in-store processor should wait until the *ready to write* acknowledgement is received via the **Ack** for the tag. A programmer can choose to use this low-level interface directly, or use various supporting hardware and software libraries provided by the platform to hide this complexity.

In our prototype implementation of minFlash and BlueDBM, we used flash chips with page sizes of 8 KB, and block sizes of 256 pages, or 2 MB. The data bus of our flash controller is 128 bits wide, so both reads and writes are done in a stream of 128 bit words.

3.1.1.2 Handling Multiple Access Agents

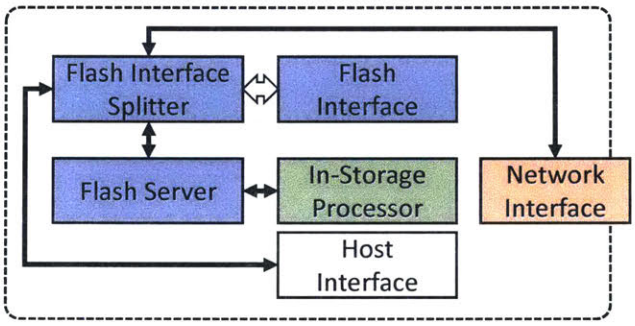


Figure 3-2: The flash interface and its surrounding components

Multiple hardware components in BlueDBM may need shared access to this flash controller interface. For example, flash storage may be accessed by local in-storage processors, local host software over PCIe DMA, or remote in-storage processors over the network. In order to simplify flash access in such a situation, we implemented a *Flash Interface Splitter* with tag renaming to manage multiple users. The flash interface splitter can be configured at compile time to have multiple interfaces with separate tag spaces, and each flash user could use its own interface as if it was using

the storage by itself. Additionally to ease development of hardware in-storage processors, we also provide an optional *Flash Server* module as part of BlueDBM. This server converts the out-of-order and interleaved flash interface into multiple simple in-order request/response interfaces by internally using page buffers. It also contains an *Address Translation Unit* that maps file handles to incoming streams of physical addresses from the host. Using these components, the in-storage processor can simply make a request with the file handle, offset and length, and the Flash Server will perform the flash operation at the corresponding physical location. The architecture of these components can be seen in Figure 3-2. The software support for this function is discussed in Section 3.2. The Flash Server can be configured at compile time based on the application's requirement, to have appropriate command queue depth, number of interfaces and the datapath width.

3.1.2 Dedicated Storage Area Network

The BlueDBM architecture includes a dedicated high-bandwidth low-latency storage area network directly between the storage devices, without having to go through the host server. The BlueDBM storage area network is a mesh network, where each node talks directly to other nodes without requiring a switch, but there is no reason it should be restricted to such an architecture. It exposes FIFO-like semantics for virtual channels to the in-storage accelerator, which assures that a single channel being blocked does not cause a network-wide deadlock. The virtual channel semantics require a transport-layer network protocol, which implemented in the hardware accelerator in order to achieve minimum latency.

The details of the network infrastructure including the protocol description and implementation are described in Chapter 4.

3.1.3 Host Interface

The in-storage processing core can be accessed from the host server over either a direct interface that supports programmed I/O as well as DMA operations, or a file system

abstraction built on top of the direct interface. This section describes the direct interface. The file system interface is implemented on top of the direct interface, and is described in detail in Section 3.2.

The host interface is implemented both on the software and hardware sides, in order to expose a more usable interface compared to a raw PCIe interface. On the software side, it exposes a queue-like interface for programmed I/O, as well as an array of page buffers for simple DMA. The software side of the host interface can be implemented either in userspace or kernelspace, depending on whether the application is a userspace application, or a kernel module such as a block device driver or a file system. An overall structure of the host interface stack can be seen in Figure 3-3.

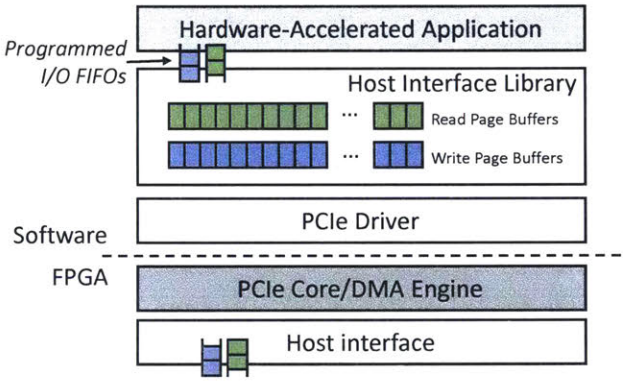


Figure 3-3: Host interface stack

In order to parallelize requests and maintain high performance, the host interface provides the software with an asynchronous I/O interface using 128 page buffers each for reads and writes. When writing a page, the user software will request a free write buffer from the host interface library, copy data to the write buffer, and send a write request to the hardware with the physical address of the destination flash page. The buffer will be returned to the free queue in the library when the hardware has finished reading the data from the buffer. When reading a page, the software will request a free read buffer from the library, and send a read request to the hardware with the physical address of the source flash page. The software can either choose to receive an interrupt or poll for the acknowledgement containing the buffer index when the hardware has finished writing to software memory.

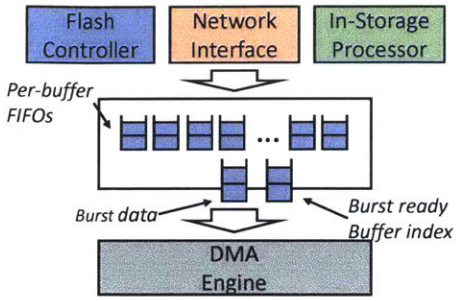


Figure 3-4: Hardware-to-host DMA burst interface over PCIe

Using DMA to write data to the storage device is straightforward to parallelize, but parallelizing reads is a bit more tricky due to the characteristics of flash storage. When writing to storage, the DMA engine on the hardware will read data from each buffer sequentially in a contiguous stream. Therefore, having enough requests in the request queue is enough to make maximum use of the host-side link bandwidth. However, data reads from flash chips on multiple buses in parallel can arrive interleaved at the DMA engine. Because the DMA engine needs to have enough contiguous data for a DMA burst before issuing it, some reordering may be required at the DMA engine. This becomes even more problematic when the device is using the storage area network to receive data from remote nodes, where they might all be coming from different buses. To fix this issue, we provide dual-ported buffer in hardware which has the semantics of a vector of FIFOs, so that data for each request can be enqueued into its own FIFO until there is enough data for a burst. Figure 3-4 describes the structure of the host interface for flash reads. Each word of data coming from any of the hardware components is tagged with a destination software buffer, and separate burst queues are maintained for each page buffer. Whenever enough data exists in any burst queue, a burst is issued to the DMA engine.

3.2 Accelerator-Aware Software Interface

In BlueDBM, we aim to provide a set of software interfaces that supports the execution of any existing application as well as modified applications that leverage the in-storage processors in the system. Furthermore, software layers in BlueDBM must perform

flash management functions since we chose to expose a raw flash interface in hardware for higher efficiency (previously discussed in Section 3.1.1). The software architecture is shown in Figure 3-5. Four types of interfaces for accessing flash storage are supplied to the user application: (1) a raw interface to flash storage through only the PCIe driver, (2) a block device driver interface, (3) a file system interface, and (4) an accelerator interface over PCIe. The first three interfaces correspond to the three leftmost arrows in Figure 3-5. The last, accelerator interface corresponds to the two rightmost arrows.

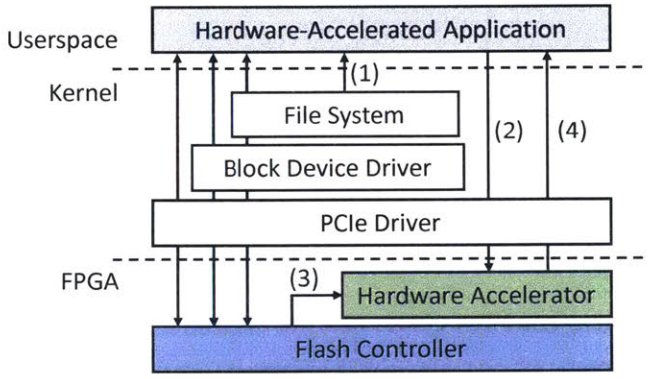


Figure 3-5: BlueDBM host interface stack

One of the major differences in the BlueDBM software interface is the lack of a default FTL in the storage device. Commercial SSDs incorporate an FTL inside the flash device controller to manage flash and maintain a block device view to the operating system. However, common file systems manage blocks in a fashion optimized for hard disks. SSDs use the FTL to emulate block device interfaces for compliance with operating systems, performing logical-to-physical mapping and garbage collection, which require large DRAM and incur lots of extra I/Os. Some file systems have tried to remedy this by refactoring the I/O architecture in order to offload most of the FTL functions into a flash-optimized log-structured file system. A prominent example of this is RFS [97] and Application-Managed Flash (AMF) [100], both of which were actually implemented and evaluated on the BlueDBM platform. Unlike conventional FTL designs where the flash characteristics are hidden from the file system, RFS and AMF performs some functionality of an FTL, including logical-to-physical address

mapping and garbage collection. This achieves better garbage collection efficiency at much lower memory requirement. The file system interface in BlueDBM is built on the same paradigm.

For compatibility with existing software, BlueDBM also offers a full-fledged FTL implemented in the device driver, similar to Fusion IO's driver. This allows us to use well-known Linux file systems (e.g., ext2/3/4) as well as database systems directly running on top of a block device with BlueDBM.

The BlueDBM software allows developers to easily make use of fast in-storage processing without any efforts to write their own custom interfaces manually. Figure 3-5 shows how user-level applications access hardware accelerators. In the BlueDBM software stack, user-level applications can query the file system for the physical locations of files on the flash (see (1) in Figure 3-5). This was made possible because the file system maintains the mapping information to the actual physical locations of the pages. Applications can then provide in-storage processors with a stream of physical addresses (see (2)), so that the in-storage processors can directly read data from flash with very low latency (see (3)). The results are sent to software memory and the user application can be notified (see (4)).

It is worth noting that in BlueDBM, all the user requests, including both user queries and data, are sent to the hardware directly, bypassing almost all of the operating system's kernel, except for essential driver modules. This helps us to avoid deep OS kernel stacks that often cause long I/O latencies. It is also very common that multiple instances of a user application may compete for the same hardware acceleration units. For efficient sharing of hardware resources, BlueDBM runs a scheduler that assigns available hardware-acceleration units to competing user-applications. In our implementation, a simple FIFO-based policy is used for request scheduling.

3.2.1 Append-Only Flash File System

One of the simplest implementations of the file systems for BlueDBM is called the Append-Only Flash File System (AOFFS). AOFFS places two restrictions on storage writes: (1) It only allows append operations to the end of files, and disallows in-

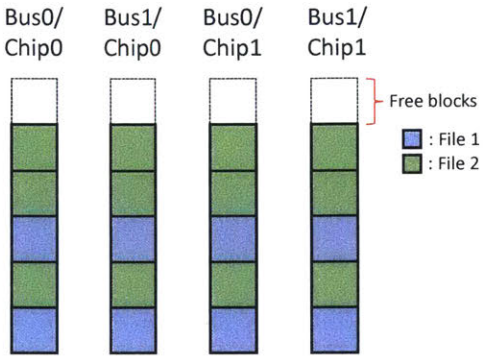


Figure 3-6: Allocation patterns of the append-only flash file system

place file updates. (2) Appends happen at very course granularities, in multiples of block sizes (multiples of 2 MB in our prototype). Page granularity reads have no restrictions, and can be read from any address.

This append-only restriction greatly simplifies flash management. A free block queue is maintained per block and chip, so that each file append can happen in a round-robin fashion across buses and chips. Internal to the allocation units, page allocation is also done in a bus and chip minor order, assuring parallelism. The multi-block allocation means there is no need for complex garbage collection. When files are erased, all blocks belonging to that file can simply be erased and added to the free block queue for its bus and chip. Furthermore, because all allocations and mappings are done in block granularity, the logical-to-physical map is also very small. An example mapping of flash blocks with two files can be seen in Figure 3-6.

For applications that only require course-grained appends and random reads, the AOFFS assures good parallelism and efficient garbage collection. One such application is the graph analytics platform described in Chapter 9.

3.2.2 Application-Specific Software Interfaces

We have also implemented various systems that expose high-level application-specific software interfaces, instead of a storage interface such as block device or file system. For example, BlueCache [185] is a distributed key-value cache appliance which plugs into the host PCIe port. It exposes to the host server the standard key-value cache

methods, using a C++ library. The interface of BlueCache is described in Table 3.2.

Method	Arguments	Returns
`set`	`char*` key, `char*` value, `size_t` key_length, `size_t` value_length	`bool`
`get`	`char*` key, `size_t` key_length, `char**` value, `size_t*` value_length	`void`
`delete`	`char*` key, `size_t` key_length	`bool`

Table 3.2: BlueCache exposes a key-value cache interface to the host

3.3 Chapter Summary

In this chapter, we have presented the architecture overview of BlueDBM, a system architecture for cost-effective analytics of Big Data using flash storage, in-store processing and hardware-accelerated storage area networks. All parts of BlueDBM, including the flash and host interfaces, the storage area network, and software interface of the in-storage accelerator, are designed to allow intelligent optimizations to achieve maximum performance from flash storage and hardware accelerators. The following sections will first introduce the design of the storage area network in detail, and then introduce the 20-node prototype implementation and its evaluation. The various applications and algorithms introduced in the following chapters have all been implemented and evaluated on the prototype BlueDBM platform.

Chapter 4

BlueDBM Storage Area Network

BlueDBM provides a low-latency high-bandwidth network infrastructure across all BlueDBM storage devices in the cluster, using a simple design with low buffer requirements. This network allows almost uniform latency access to the entire flash array, allowing for a uniform address space abstraction.

BlueDBM storage devices are networked among themselves via high-performance serial links, separate from the network connecting the host machines together. The BlueDBM network is a packet-switched mesh network, in which each storage device has multiple network ports and is capable of routing packets across the network without requiring a separate switch or router. In addition to routing, the storage network supports functionality such as flow control and virtual channels while maintaining high performance and extremely low latency. For data traffic between the storage devices, the storage area network ports remove the overhead of going to the host software to access a separate network interface.

This chapter is organized as follows: Section 4.1 introduces the overall architecture and goal of the storage area network, and Section 4.2 describes the design and implementation of the various protocol layers in the network. Section 4.3 concludes the chapter.

4.1 Network Architecture Overview

Figure 4-1 shows the network architecture. Switching is done at two levels: the internal switch and the external switch. The internal switch routes packets between local components. The external switch accesses multiple physical network ports, and is responsible for forwarding data from one port to another in order to relay a packet to its next hop. It is also responsible for relaying inbound packets to the internal switch, and relaying outbound packets from the internal switch to a correct physical port.

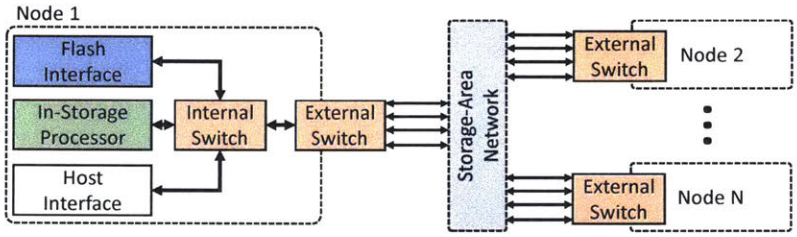
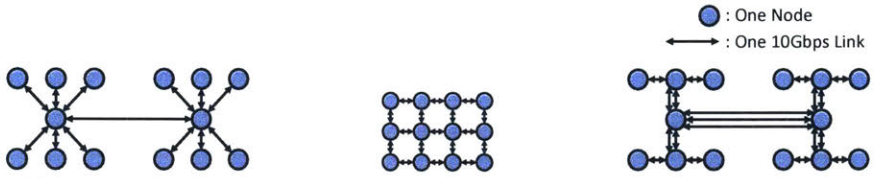


Figure 4-1: Network architecture

Due to the multiple network ports on the storage nodes, the BlueDBM network is very flexible and can be configured to implement various topologies, as long as there is sufficient number of ports on each node. Figure 4-2 shows some examples of topologies. To implement a different topology, the physical cables between each node have to be re-wired, but the routing across a topology can be configured dynamically by the software.



(a) Distributed star (b) Mesh (c) Fat tree

Figure 4-2: Any network topology is possible as long as it requires less than the available number of network ports per node

The storage-area network implements transport-layer semantics, according to the OSI 7-layer model, to the in-storage processor. The in-storage processor can instan-

tiate multiple endpoints at compile time, and each endpoint can communicate with corresponding endpoints in remote storage devices. The network infrastructure supports virtual channels between corresponding endpoints, implementing end-to-end flow control to ensure lossless communication, and that blocking in one channel will not cause deadlocks in others.

4.1.1 Logical Endpoint Network Interface

The BlueDBM network infrastructure exposes virtual channel semantics to the users of the network by providing them with multiple *logical endpoints*. The number of endpoints is determined at design time by setting a parameter, and all endpoints share the physical network. Each endpoint is parameterized with a unique index that does not need to be contiguous. Each endpoint exposes two interfaces, `send(dst,data)` and `receive`, and can be used as if using a FIFO. An in-storage processor can send data to a remote node by calling `send` with a pair of data and destination node index, or receive data from remote nodes by calling `receive`, which returns a pair of data and source node index. End-to-end flow control is implemented inside the endpoints, so the network between the local endpoints can simply relay packets without worrying about deadlocks. The internal architecture of the logical endpoint that implements transport layer semantics such as virtual channels and end-to-end flow control is described in Section 4.2.3. Such intuitive characteristics of the network ease development of in-storage processors. Figure 4-3 shows how endpoints can be connected to the network interface.

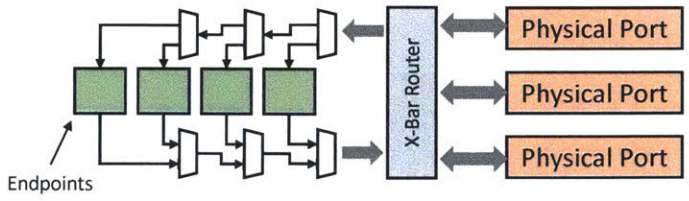


Figure 4-3: Network architecture with Logical Endpoints

4.2 Protocol Design and Implementation

The storage area network implements transport layer semantics, so that each logical endpoint behaves like a pair of FIFOs, each for sending and receiving.

4.2.1 Link Layer

The link layer implementation of the BlueDBM storage area network manages physical connections between network ports in the storage nodes. The most important aspect of the link layer is the simple token-based flow control implementation. This provides back pressure across the link and ensures that packets will not drop if the data rate is higher than what the network can manage, or if the data cannot be received by the destination node which is running slowly.

4.2.2 Network Layer

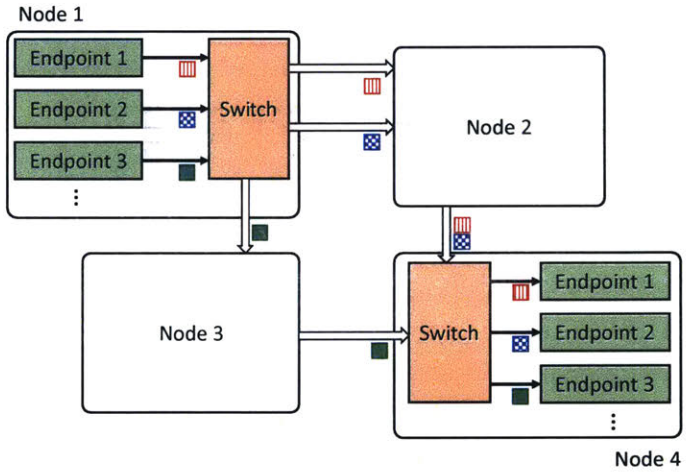


Figure 4-4: Packets from the same Endpoint to a destination maintain FIFO order

In order to make maximum use of the bandwidth of the network infrastructure while keeping resource usage to a minimal, the BlueDBM network implements deterministic routing for each logical endpoint. This means that all packets originating from the same logical endpoint that are directed to the same destination node follow the same route across the network, while packets from a different endpoint directed

to the same destination node may follow a different path. Figure 4-4 shows packet routing in an example network. The benefit of this approach is that packet traffic can be distributed across multiple links, while maintaining the order of all packets from the same endpoint. If packets from the same endpoint are allowed to take different paths, it would require a completion buffer which may be expensive in an embedded system. For simplicity, the BlueDBM network does not implement a discovery protocol, and relies on a network configuration file to populate the routing tables.

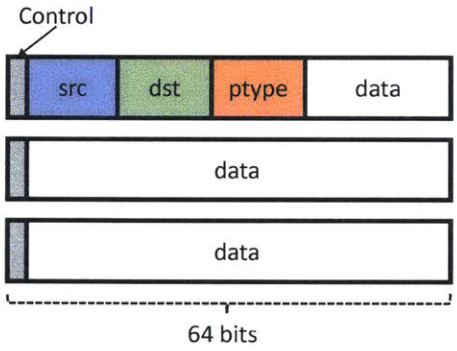


Figure 4-5: Data layout of a packet

The internal structure of a BlueDBM network packet can be seen in Figure 4-5. In network nomenclature, a unit of data transferred each cycle is called a *phit*, or a physical unit. Given a fixed-width network link (e.g., 64 bits), each packet can be transferred over multiple cycles, in multiple phits. For simplicity, each phit is tagged with a control field, which determines if the phit is a link, network or transport-level flow control packet, or whether it is a header or payload phit.

4.2.3 Transport Layer

Virtual channels multiplex a single physical network link to provide the logical interface of multiple links. Our network implements a per-channel end-to-end flow control, so that a sender can only send data onto the network when it is guaranteed that the receiving endpoint has enough buffer space to accommodate it. The transport layer is implemented in individual endpoints, and its design aims to provide a very low latency and efficient memory space usage. Each design can have multiple instantiations

of endpoints, parameterized differently.

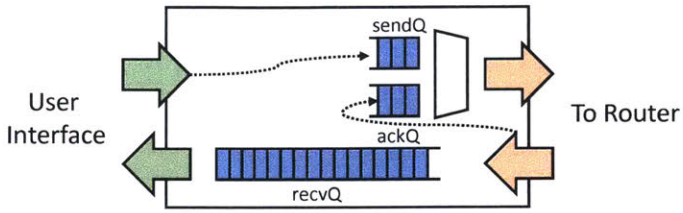


Figure 4-6: End-to-end flow control is implemented in the endpoint

The structure of an endpoint is described in Figure 4-6. Whenever a packet is received by an endpoint, it checks a table of the numbers packets received per source node to determine if it is time to send a flow control credit to the source node. If the send budget of the source node is predicted to have become small enough and there is enough space on the local receive buffer, it enters a packet into the ack queue and marks the amount of space as allocated.

In order to maintain maximum bandwidth, flow control packets must be received before the send budget of the source node runs out. However, it is often not possible to provide a large enough buffer to conservatively accommodate the flow control packet's round trip latency for all nodes in the system.

Under such constraints, each endpoint can have a different flow control configuration that attempts to best suit its usage. For example, for some endpoints the expected traffic pattern may be that most of the data transfer happens between a pair of two nodes. In such a case, it might be effective to have a very low granularity flow control, so that a large buffer is allocated on request, but the total receive buffer may be small. On the other hand, if many nodes are expected to send data to one node, a fine granularity flow control and a large buffer may be required for performance. If the endpoint is used for low-bandwidth traffic such as commands, the buffer size and granularity can be set to a small value. To enable such control, endpoints are initialized with the parameters described in Table 4.1.

Initially, all nodes start with a small send budget (`initBudget`) to all remote nodes, and therefore the actual size of the receive packet buffer is $initBudget \times nodeCount$ slots larger than the `BufferSize` parameter. When the first packet ar-

Parameter	Description
BufferSize	Size of the total allocated buffer space
FlowOffset	Offset of flow control packet transmission
FlowCredit	Number of packets each flow control credit represents

Table 4.1: Network endpoint parameters

rives, space in the receive buffer is allocated to the source node and a flow control packet is sent. The endpoint can choose to periodically allocate only `initBudget` size buffers, instead of `FlowCredit` in an attempt to more fairly allocate buffer space across source nodes. Yielding buffers like this achieves better buffer usage when many nodes are going to send data to one node.

The network infrastructure also provides an unmanaged endpoint which does not implement a transport layer protocol. The unmanaged endpoint can sustain the highest bandwidth and the lowest latency as it does not check if the receiving endpoint has available buffers before sending packets. It should be used very carefully, since if the arriving data is not always immediately consumed and dequeued from the receiving buffer, it may cause the entire network to be blocked.

4.3 Chapter Summary

This chapter introduced the design of the BlueDBM storage area network, which provides parameterized FIFO-like virtual channel semantics to the accelerators implemented in the in-storage accelerator. All characteristics of the storage area network were designed to have low buffer size and circuit overhead in hardware, and to achieve minimum latency. The performance of the storage area network will be presented in Chapter 5, using the 20-node prototype implementation of BlueDBM and the network.

Chapter 5

BlueDBM Prototype Cluster

This chapter describes the 20-node BlueDBM cluster with 20 TB of flash storage that we have built to explore the capabilities of the architecture described in the earlier chapters. Figure 5-1 shows a photo of our implementation.

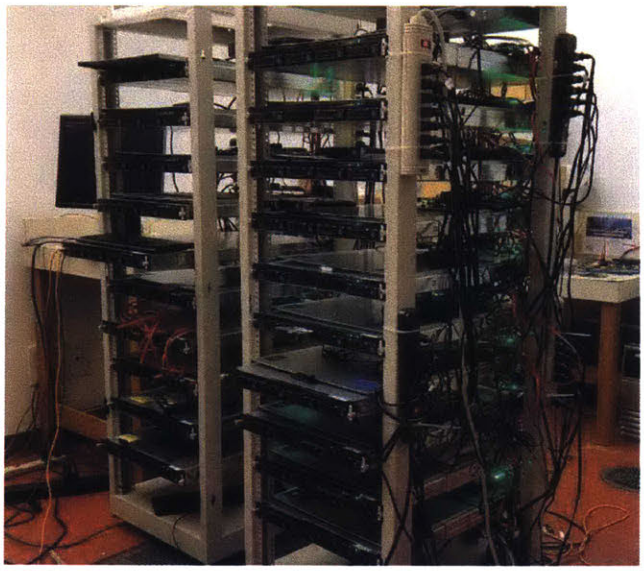


Figure 5-1: A 20-node BlueDBM prototype cluster

This chapter is organized as follows: Section 5.1 describes the implementation details of the 20-node prototype. Section 5.2 introduces the results from performance and other benchmarks we have run to evaluate the prototype cluster. Section 5.3 shows the evaluation results from a graph traversal benchmark, a simple latency-

sensitive application. Section 5.4 summarizes this chapter.

5.1 Prototype Implementation

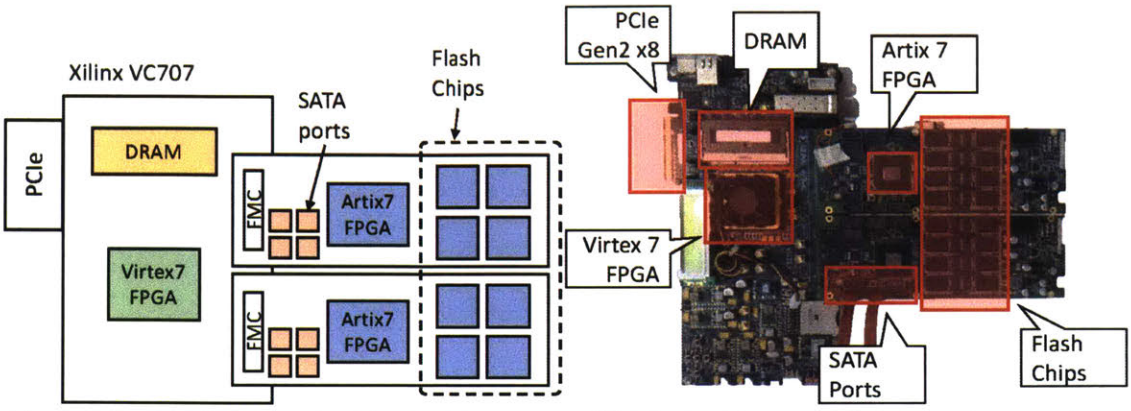
In our implementation of BlueDBM, we have used a Field Programmable Gate Array (FPGA) to implement the in-storage processor and also the flash, host and network controllers. However, the BlueDBM Architecture should not be limited to an FPGA-based implementation. Development of BlueDBM was done in the high-level hardware description language Bluespec. It is possible to develop in-storage processors in any hardware description language, as long as it conforms to the interface exposed by the BlueDBM system services. Most of the interfaces are latency-insensitive FIFOs with backpressure. Bluespec provides a great amount of support for such interfaces, making in-storage accelerator development easier.

The prototype cluster consists of 20 rack-mounted Xeon servers, each with 24 cores and 50 GB of DRAM. A BlueDBM storage device is plugged into each server via a PCIe connection. A BlueDBM storage device consists of a Xilinx VC707 FPGA development board, augmented with two custom flash boards plugged into each of its two FPGA Mezzanine Card (FMC) ports. The VC707 FPGA development board includes a 1 GB DDR3 DRAM card. Each custom flash board also has multiple SATA connectors, which are not used for normal SATA storage device protocol, but instead used for the dedicated storage area network. The SATA form factor was chosen because SATA cables were easier to manage than coaxial SMA cables. Figure 5-2 shows the components of a single BlueDBM storage device.

5.1.1 minFlash Custom Flash Card

We have designed and built a high-capacity custom flash board called minFlash with high-speed serial connectors, with the help of Quanta Inc., and Xilinx Inc. Its physical implementation was shown in Figure 1-2. Each BlueDBM storage device includes two minFlash cards, adding up to 1 TB of flash storage for each node.

Each flash card consists of sixteen 256 Gb NAND flash chips organized into 8



(a) Component diagram of a BlueDBM storage device

(b) BlueDBM storage device implementation

Figure 5-2: A BlueDBM storage device

buses of 2 chips each, adding up to 512 GB of storage per card. The maximum internal bandwidth of each custom flash card is 1.2 GB/s, adding up to 2.4 GB/s of bandwidth across both flash cards. This maximum bandwidth can be achieved if all buses and chips are kept busy, regardless of whether the access pattern is sequential or random.

Each flash card also carries a Xilinx Artix 7 chip, where the chip-specific flash controller and error-correcting code (ECC) are implemented, providing the Virtex 7 FPGA chip on the VC707 a logical error-free access into flash. In the interest of simplicity, we have implemented RS(255,243) Reed-Solomon codes for the ECC implemented in the Artix 7 chip. The communication between the Artix 7 and the Virtex 7 FPGA is done over a 4-lane serial channel, which is implemented using the GTX/GTP serial transceivers included in each FPGA, and uses the Xilinx Aurora link-level protocol. This channel can sustain up to 3.3 GB/s of bandwidth at 0.5 μs latency, which exceeds the bandwidth of the flash chips. The flash board also hosts 8 SATA connectors, 4 of which pin out the high-speed serial ports on the host Virtex 7 FPGA, and 4 of whch pin out the high-speed serial ports on the Artix 7 chip. The serial ports are capable of 10 Gbps and 6.6 Gbps of bandwidth, respectively. A total of eight 10 Gbps connectors on the two flash cards is used to create the storage area mesh network between the in-storage processors.

5.1.2 Storage-Area Network Infrastructure

As mentioned in the previous section, the network fabric used for the dedicated storage-area network uses the 10 Gbps serial connections available in the FPGA. The latency of this fabric, as well as the Xilinx Aurora link-level network protocol, is very low. And by implementing a latency-sensitive network protocol inside the hardware, we were able to achieve very high performance and low latency communication between the in-storage processors.

Each channel in our network infrastructure consisted of two 10 Gbps connectors, adding up to 20 Gbps in total per channel. After the protocol overhead, we were able to measure 17 Gbps of effective bandwidth for each link, at less than 0.5 μs latency per network hop. By using two links per channel, our implementation has a network fan-out of 4 ports per storage device, which can be used to create a wide variety of network topologies.

5.2 Performance Evaluation

In this section, we present some low-level performance numbers collected from the BlueDBM prototype cluster, including the network performance, distributed storage access latency and bandwidth, as well as estimated power consumption numbers.

5.2.1 Network Performance

5.2.1.1 Simple Bandwidth Evaluation

We measured the performance of the network by transferring a single stream of 128 bit data packets through multiple nodes across the network in a non-contentious traffic setting. The maximum physical link bandwidth is 20 Gbps, and per-hop latency is 0.48 μs. Figure 5-3 demonstrates that we are able to sustain 17 Gbps of bandwidth per stream across multiple network hops. This shows that the protocol overhead is under 15%. The latency is 0.48 μs per network hop, the end-to-end latency is simply a multiple of network hops to the destination.

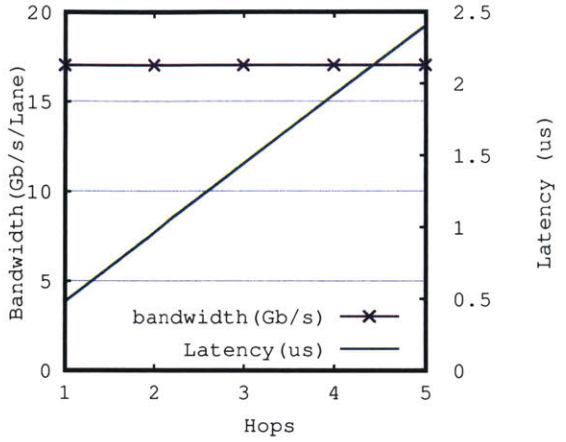


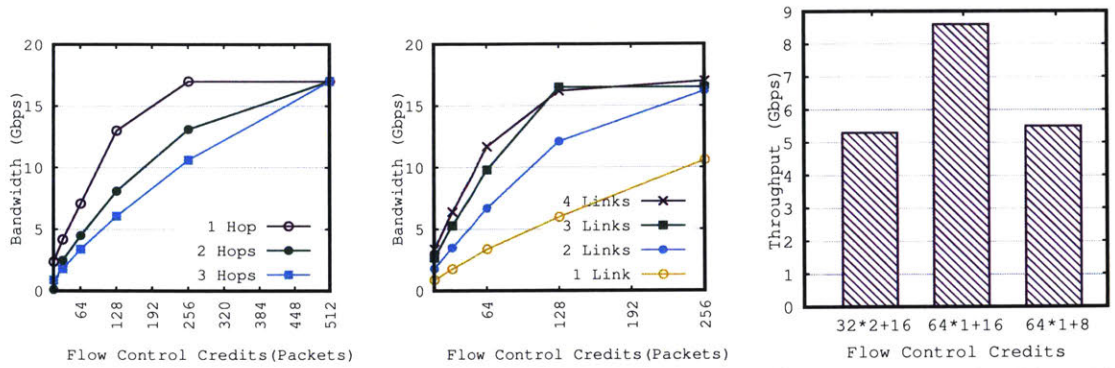
Figure 5-3: BlueDBM storage area network performance

Each node in our BlueDBM implementation includes a fan-out of 4 network channels, so each node can have an aggregate full duplex bandwidth of 8.5 GB/s. With such a high fan-out, it would be unlikely that a remote node in a rack-class cluster to be over 4 hops, or 2 μs away. In a naive ring network of 20 nodes with 4 lanes each to next and previous nodes, the average latency to a remote node is 5 hops, or 2.5 μs. The ring throughput would be 32.8 Gbps. *Assuming a flash access latency of 50 μs, such a network will only add 5% latency in the worst case, giving the illusion of a uniform access storage.*

5.2.1.2 Protocol Performance Evaluation

We demonstrate that the performance of the network does not suffer from the addition of transport-layer network functions. We measured the bandwidth and latency of the network under various configurations, and show that our network can usually achieve a bandwidth of 17 Gb/s, which is 85% of the maximum physical link bandwidth. This performance is reasonable considering the packet header and flow control overhead.

Single Endpoint Over Multiple Hops We measured the bandwidth of the network implementation by measuring the time it takes for a single endpoint to send a large amount of data to remote nodes variable hops away, under various flow control settings. Larger credit settings mean that a larger buffer size is required. Flow control



(a) Network bandwidth with variable network distance

(b) Network bandwidth with variable number of channels

(c) Network bandwidth with different flow control settings

Figure 5-4: Performance evaluation of the transport-layer protocol implementation

offset was set to be half of the credit size. The results can be seen in Figure 5-4a.

When the flow control credit was small, performance of the network was lower when going over a longer network distance. This is because the round trip latency over multiple hops is longer than the time it takes to deplete the send buffer, resulting in idle cycles when no data can be safely sent over the network. With the low network latency of the serial links, maximum bandwidth over 3 network hops could be achieved using a single endpoint when the flow control credit is over 512 packets large.

Multiple Endpoints Over Multiple Hops Since most interesting distributed FPGA applications will have more than one network endpoint, maximum network performance can be achieved even when a single endpoint's flow control credit setting is large enough. We measured the aggregate network bandwidth of a varying number of endpoints sending data to a node three network hops away. We also measured the performance with varying flow control credit sizes. Flow control offset was set to be half of the credit size. The results can be seen in Figure 5-4b. It shows that a collection of smaller sized endpoints can saturate the network by filling in each other's idle cycles.

Buffer Size and Flow Control Offset Endpoints can be characterized not only by its flow control credit size, but also by the flow control offset and buffer size

parameters. The same amount of buffer space can also be allocated to a different number of nodes under different flow control credit settings. Setting a smaller offset has the risk of incurring idle time by delivering a flow control packet too late, but a large offset requires a larger buffer to accommodate earlier buffer allocation.

We measured the effect of such parameters by having three nodes send a stream of packets to the same remote node. We tested three scenarios, described in Table 5.1. Two had the same total buffer size organized into different organizations, and one had a smaller buffer. In the first scenario, the three source nodes will be contending to be scheduled into the two possible slots, where in the latter two scenarios they will be contending for one 64 packet slot.

Setting	Description
32*2+16	Buffer has space for two 32 packet blocks, with offset of 16
64*1+16	Buffer has space for one 64 packet block, with offset of 16
64*1+8	Buffer has space for one 64 packet block, with offset of 8

Table 5.1: Flow control parameters

The results can be seen in Figure 5-4c. It shows that even with the same buffer size, having a larger credit setting is beneficial compared to a small buffer configuration. The difference is pronounced enough that even reducing buffer usage further by making the offset smaller results in a better performance compared to the configuration with smaller credit sizes.

5.2.2 Remote Storage Access Latency

We measured the latency of remote storage access by reading an 8 KB page of data from various sources using the dedicated storage area network. The list of data sources are shown in Table 5.2.

In each case, the request is sent from either the host server or the in-store processor on the local BlueDBM node. For the sake of fairness, all network communications in all experiments were done over the storage area network, using its RDMA features to reduce latency. In the third and fourth cases, the request is processed by the remote server, instead of the remote in-store processor, adding extra latency. However, even

Name	Description
ISP-F	From in-store processor to remote flash storage
CPU-F	From host server to remote flash storage
CPU-N-F	From host server to remote flash storage via its host server
CPU-M	From host server to remote DRAM

Table 5.2: Data sources for measuring remote access latency

when a request is processed by the remote server, data is always transferred back via the storage area network. We could have also measured the accesses to remote servers via Ethernet, but that latency is at least 100x of the storage area network, and will not be particularly illuminating.

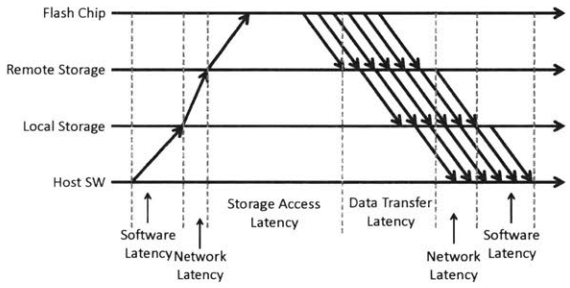


Figure 5-5: Breakdown of remote storage access latency

The latency is broken up into four components as shown in Figure 5-5. The first is the local software overhead of accessing the network interface. The second is the storage access latency, or the time it takes for the first data byte to come out of the storage device. The third is the amount of time it takes to transfer the data until the last byte is sent back over the network, and the last is the network latency.

Figure 5-6 shows the exact latency breakdown for each experiment. Notice in all 4 cases, the network latency is insignificant. The data transfer latency is similar except when data is transferred from DRAM (CPU-M), where it is slightly lower. Except in the case of ISP-F, storage access incurs the additional overhead of PCIe and host software latencies. If we compare ISP-F to CPU-N-F, we can see the benefits of a storage area network, as the former allows overlapping the latencies of storage and network access.

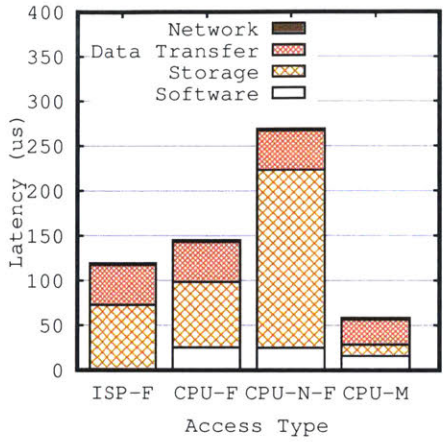


Figure 5-6: Latency of remote data access in BlueDBM

5.2.3 Storage Access Bandwidth

We measured the bandwidth of BlueDBM by sending a stream of millions of sequential read requests for 8 KB size pages to local and remote storage nodes, and measuring the elapsed time for the requesting node's in-storage processor to finish receiving all data. We tested the bandwidth available to a single requesting node in a cluster of 1 to 4 nodes. The cluster was configured to have a fully connected topology, using the standard 2-lane channels between individual nodes.

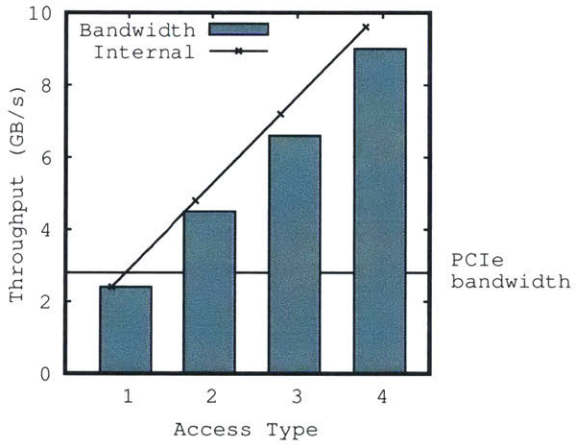


Figure 5-7: Bandwidth available to a single in-storage processor in a cluster

Figure 5-7 shows the performance available to a single node, compared to the total internal bandwidth that should be available to the cluster. While the total bandwidth available to the requesting node is scalable with a larger cluster, it is not

fully saturating the total available internal flash bandwidth of the cluster. This is because a single channel of the storage-area network, which achieves about 1.7 Gbps of bandwidth, is becoming the bottleneck, compared to the 19.2 Gbps per node bandwidth of flash storage. The graph also shows that as the bandwidth of multiple nodes is focused into a single device, the PCIe bandwidth, even at Gen2 x8 performance, quickly becomes the bottleneck. This emphasizes the benefits of using an in-storage accelerator near storage, because it is not bound by the hostside link bandwidth, and can consume the entire bandwidth if it has enough computation capacity. Furthermore, the results in Figure 5-7 are collected from a cluster with a fully connected network topology, which is not possible with a cluster size of more than 5 nodes. As cluster sizes become larger, the network bandwidth can also become a bottleneck. In such a situation, in-storage computation becomes critical to make the best use of available flash bandwidth, because it may be able to reduce the amount of data over the network by performing filtering or pre-processing.

5.2.4 Power Consumption

Table 5.3 shows the overall power consumption of the system, which was estimated using values from the datasheet. Each Xeon server includes 24 cores and 48 GB of DRAM, and power consumption was measured while all cores were busy with a bandwidth-stressing micro-benchmark. Thanks to the low power consumption of the FPGA and flash devices, BlueDBM adds less than 20% of power consumption to the system.

Component	Power (Watts)
VC707	30
Flash Board x2	10
Xeon Server (Busy)	240
Node Total	280

Table 5.3: BlueDBM estimated power consumption

5.3 Latency-Sensitive Application Results: Graph Traversal

5.3.1 Application Description

Efficient graph traversal is a very important component of any graph processing system. Fast graph traversal enables solving many problems in graph theory, including depth-first search, loop detection, shortest path, and maximum flow. It is also a very latency-bound problem because one often cannot predict the next node to visit, until the previous node is visited and processed. We demonstrate the performance benefits of our BlueDBM architecture by implementing distributed graph traversal that takes advantages of the in-store processor and the storage area network, which allows extremely low-latency access into both local and remote flash storage.

5.3.2 Performance Evaluation

Graph traversal algorithms often involve dependent lookups. That is, the data from the first request determines the next request, like a linked-list traversal at the page level. Since such traversals are very sensitive to latency, we conducted the experiments with settings that are very similar to the settings in Section 5.2.2. These settings can be seen in Table 5.4.

Figure 5-8 shows the performance evaluation results. As expected, the results show that the storage area network and in-store processor together exhibit almost a factor of 3 performance improvement over a generic distributed SSD. This performance difference is large enough that even when 50% of the accesses can be accommodated by DRAM, performance of BlueDBM is still much higher. This result is interesting because a system with enough DRAM to accommodate half the graph accesses in memory would most likely be costlier than a fully flash-based system. Similarly, while the performance of **H-DRAM** is still double that of **ISP-F**, it should be noted that a distributed system with enough DRAM to accommodate the entire graph structure would most likely cost much more than double that of a system with enough flash

Setting	Description
ISP-F	In-store processor requests data from remote storage over storage area network
CPU-F	Software requests data from remote storage over storage area network
CPU-N-F	Software requests data from remote software to read from flash
DRAM + 50% F	Software requests data from remote software. 50% chance of hitting flash
DRAM + 30% F	Software requests data from remote software. 30% chance of hitting flash
H-DRAM	Software requests data from remote software. Data read from DRAM

Table 5.4: System configurations for benchmarking the graph traversal application

storage.

The performance difference between **CPU-F** and **CPU-N-F** illustrates the benefits of using the storage area network to reduce a layer of software access. Performance of **ISP-F** compared to **CPU-F** shows the benefits of further reducing software overhead by having the ISP manage the graph traversal logic.

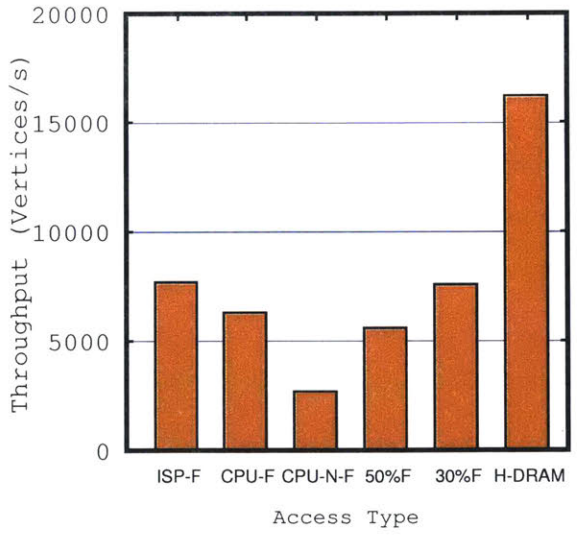


Figure 5-8: Performance evaluation of the graph traversal application

5.4 Chapter Summary

This chapter described the 20-node prototype implementation of BlueDBM, and presented results of some simple performance benchmarks to demonstrate all system components are functioning at high efficiency. The low latency of remote storage access enabled by the storage area network and the in-storage accelerator helped achieve relatively high performance for latency-bound problems like graph traversal, in which a fully flash-based system could outperform a much costlier system with enough DRAM to store up to 70% of the data in DRAM. The following chapters explore various applications implemented on the BlueDBM platform.

Part II

Application Evaluations

Chapter 6

Approximate High-Dimensional Nearest-Neighbor Search

Modern datasets of importance such as images, videos, protein sequences or text, usually contain very high dimensional information from the search point of view. Nearest-neighbor search is one of the most fundamental building blocks in dealing with large amounts of data. It is the problem of finding points in a database that are most similar to a query data point by some distance metric. Since it is difficult to pre-process high dimensional datasets and structure them into an easily queried format, performance-sensitive systems often use approximate search algorithms, which may be random access-intensive. Thanks to the high dimensionality of data resulting in large size of data elements, this application does not suffer from the coarse access granularity of flash storage. Also, in terms of storage performance, it is mostly a bandwidth-bound problem, and not sensitive to access latency.

Our evaluations show that for even moderately complex distance metrics, a hardware-accelerated flash-based implementation can perform on par or beyond an in-memory software system, as the computational overhead becomes the bottleneck. Even for very simple metrics, a flash-based system can outperform an in-memory system where even a fraction of data does not fit completely in memory, as the performance drop is steep.

This chapter is organized as follows: Section 6.1 introduces background and ex-

isting research related to high-dimensional nearest-neighbor search. Section 6.2 describes the system architecture and implementation information of the system. Section 6.3 provides performance evaluation results of our system. We summarize and conclude in Section 6.4.

6.1 Background

6.1.1 High-Dimensional Nearest-Neighbor Search

Nearest-neighbor search on high-dimensional data suffers from a so-called "curse of dimensionality," in which it becomes very difficult to pre-process data and organize them it a structure that is easily queried. For example, kd-trees [22] are often used for low-dimensional nearest-neighbor queries, but become less effective in a high dimensional setting because the relative importance of any single dimension becomes low. In the worst case, a query has to iteratively compute the distance function between the query point and all points in the dataset to determine the nearest neighbor. This becomes especially difficult since distance calculation often does not benefit from preprocessing and requires comparing two raw data points when the dimensionality is high. A good example of this is time series comparisons.

Many clever indexing and sampling schemes have attempted to solve this problem [128, 117]. A prominent work in this area is Locality Sensitive Hashing (LSH) [58], where the dataset is pre-processed using a set of hash functions, and each element is assigned to one of many buckets. The same hash functions are applied to a query, and only the data already in the corresponding bucket has to be compared. LSH is an approximate nearest-neighbor algorithm, where it provides some statistical guarantee of finding the nearest neighbor. An interesting aspect of such algorithms is that data is no longer read in a sequential manner, because data in a certain bucket may be scattered across the dataset, as seen in Figure 6-1. As a result, random access performance of the storage device becomes a limiting factor of performance. Because random access performance of hard disk drives is very bad due to its high seek time,

performance often drops sharply when data does not fit completely in DRAM. For in-memory systems, the large size of modern datasets requires data and computation to be distributed across many machines in a cluster, using distributed processing platforms such as Hadoop [174] or Spark [188].

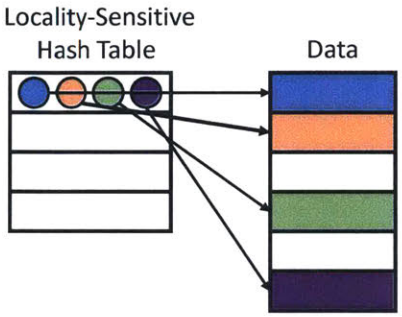


Figure 6-1: Data accesses in LSH are not sequential

6.1.2 Comparing Images

A popular application of nearest-neighbor search is content-based image retrieval, where a system looks for images that are most similar to a query image, according to some distance metric. The usefulness of a content-based image retrieval system is determined not only by how fast the result can be obtained, but also by the usefulness of the distance metric. There exist image comparison techniques optimized for various applications, including feature extraction [111, 20], facial recognition [154] and scene recognition [191].

Histograms using just color values of individual pixels (RGB or HSV) have been useful in the past, as they often correctly represent the color profile of the images while being robust in regards to movements in the image such as shifting. However, color histograms begin to fail when the image corpus becomes large, when there are different images that have similar color compositions. Joint histograms [144] overcome this limitation by adding more features into consideration. A joint histogram of an image is a multidimensional histogram built using color values of each pixel and other local pixel features, such as edge density, texturedness and gradient magnitude of a pixel location. In constructing joint histograms, an image is first processed using

various filters that generate new images that emphasize different local features of the image. A histogram constructed using these images in addition to color values is much more representative of the features of the image.

6.1.3 Document Similarity Search

Another important application where nearest-neighbor search is employed is document similarity search, where a database of documents is searched to retrieve those that are similar to a query document. Document similarity search is used in applications such as finding similar web pages on the Internet, plagiarism detection or searching for relevant legal documentation. A popular method of measuring similarity between documents is term similarity, where documents are deemed similar if they share the same terms, or words. In term similarity comparison, a document is viewed as a "bag of words," where a document is characterized by the orderless set of words and the number of occurrence of each word. The simplest way to compare two bags of words is to count the occurrence of shared words. However, this simple method often does not perform well in terms of accuracy. A simple yet effective method is using *Cosine Similarity*, where each bag of words is considered a multidimensional vector, where each term corresponds to a dimension and its occurrence corresponds to the magnitude. Similarity between vectors is determined by calculating the cosine between the two vectors. In this work, we have created a synthetic database of documents pre-processed into bags of words, and performed nearest-neighbor search using cosine similarity as the distance metric.

6.1.4 Neural Networks and Nearest-Neighbor Search

A modern way of image or document retrieval that has quickly gained popularity is using deep neural networks [94]. Each image is processed using a neural network and is given a set of tags that are determined relevant to the image, reducing the dimensionality by a great amount. Using deep neural networks to classify images has shown enormous accuracy, and is becoming the de facto method of image or

document classification. In a neural network-based image or document search, images or documents would be pre-processed during ingestion and augmented with metadata, such as descriptive tags or histogram values. The process of searching a dataset consisting of such metadata would be very similar to our document search scenario.

6.2 Architecture and Implementation

Each in-storage processor implements an array of application-specific distance comparators and a controller that manages them. Locality-sensitive hashing is implemented in the host software, and the host software tells the in-storage accelerator which locations need to be read and compared. The calculated distance from the accelerator is read by the host software, and it keeps track of a certain number of data points that are the most similar to the query. The in-storage accelerator sends read commands to the flash array to read individual data points, and routes the data to idle comparators.

We compared the performance of various system configurations using distance metrics of variable complexity. The distance metrics we tested are presented in Table 6.1.

Configuration	Complexity
Bitwise Hamming Distance	Low
Cosine Similarity	Medium
Image Comparison	High

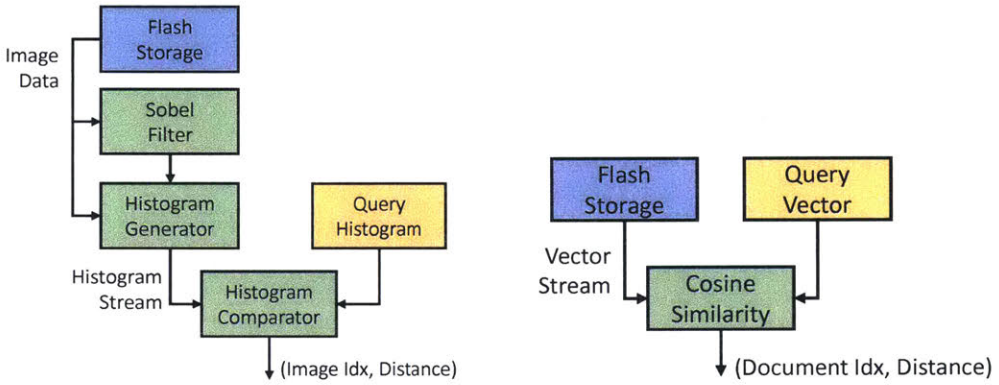
Table 6.1: System configurations for nearest-neighbor search

6.2.1 Image Comparator Architecture

Figure 6-2a shows the internal structure of the in-storage processor for content-based image comparison using joint histograms. The comparator reads a stream of RGB-encoded fixed-size image data and emits its distance against the query image, using the calculated histogram of the query image. It uses each pixel's RGB values as

well as its *edgeness*, calculated by a sobel edge detection filter, in order to create a $4 \times 4 \times 4 \times 4$ histogram.

While all accelerator components in the image comparator, including the sobel filter, are pipelined, resulting in wire-speed performance, the 3 byte-per-cycle datapath (RGB values) was not wide enough for a single comparator to saturate the flash storage. We instantiated 16 accelerators that are assigned work in a round-robin fashion in order to completely saturate flash bandwidth in all circumstances.



(a) Internal architecture of the content-based image comparator

(b) Internal architecture of the document comparator

Figure 6-2: Architecture of image and document distance calculators

6.2.2 Document Comparator Architecture

Figure 6-2b shows the internal structure of the document comparator accelerator. The cosine similarity-based document comparator is much simpler and less computation-heavy than the image comparator. Each document in the dataset is pre-processed and stored as a vector of tuples consisting of a word and the number of its occurrences. The vocabulary is determined beforehand, so each document is represented using a fixed-length vector of numbers representing the occurrence of each word. The comparator reads a stream of encoded document vectors and calculates the cosine similarity against the query vector. As it is trivial to extract internal parallelism from cosine similarity, a single comparator can have a wide enough datapath to saturate flash storage.

The architecture of the hamming distance comparator is almost identical to the document comparator, but the cosine similarity calculator is replaced with a hamming distance calculator.

6.3 Performance Evaluation

6.3.1 Evaluation with Different Metric Complexity

We tested the performance of the three distance metrics using the system configurations in Table 6.2. Figure 6-3 shows the collected results, normalized against the DRAM-based system, which is the upper bound of performance for systems without hardware acceleration. Locality sensitive hashing was configured with a bucket count of 30, which is not a very large number, but still helps illustrate the impact of random accesses without requiring too large datasets.

For the image dataset, we used the ImageNet [45] dataset, resized to have width and height of 128 pixels. The resized dataset had a capacity of roughly 1TB, and fit completely in two nodes. We used a synthetic document dataset using English words for the document dataset, as well as the hamming distance dataset. Each document was pre-processed into a vector of 8000 8-bit values, each representing the occurrence of a word in the document.

Configuration	Description
Random Disk+SW	Data stored on disk, processed using 24-thread software
BlueDBM	Data stored on BlueDBM, processed using hardware
DRAM+SW	Data stored completely on DRAM, processed using 24-thread software

Table 6.2: System configurations for nearest-neighbor search

As distance metrics become more complex, the relative performance of the disk-based system and BlueDBM increases, as increasing amount of time is spent on processing the data, rather than fetching it from storage. Due to terrible random access performance of hard disks, the disk-based system shows relatively low performance even with complex metrics. However, BlueDBM's fast flash storage, coupled with

high-performance distance comparator engines that fully saturate storage bandwidth, allows examples with even a relatively simple distance metric such as cosine similarity to show performance rivalling the DRAM-based system's performance. With a more complex distance metric such as image comparison, the computation overhead becomes a bigger bottleneck than flash storage performance. In such a situation, fast flash storage with a fast hardware accelerator near the storage can exceed the performance of a DRAM-based system. In all situations, the disk-based system performed extremely poorly due to its bad random access performance. We think the examples shown here cover a wide range of meaningful applications, and shows that BlueDBM can be a desirable platform for accelerating this class of applications.

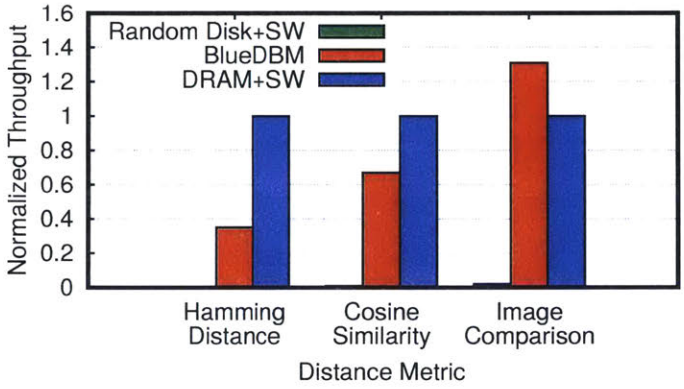


Figure 6-3: BlueDBM's relative performance increases with more complex distance metrics

6.3.2 Evaluation with Different DRAM Coverage

Another interesting observation from the nearest-neighbor application was that performance of a DRAM-based system with backing secondary storage drops sharply when even a small amount of requests overflows from memory into secondary storage via swapping. We tested the performance of various system configurations running LSH-based approximate nearest search, using the simplest hamming distance metric. We tested the throughput of the system configurations in Table 6.3.

Figure 6-4 shows the performance of these configurations. Even when 90% of accesses is contained in memory, and only 10% of accesses spills over into fast flash

Configuration	Description
DRAM	System with enough DRAM to cover all read requests
ISP	BlueDBM system with in-storage hamming distance accelerator
10% Flash	System where 10% of reads spill over into flash storage
Full Flash	System where 100% of reads are from flash storage
5% Disk	System where 5% of reads spill over into disk

Table 6.3: System configurations with different DRAM coverage

storage, performance drops sharply to the point it is comparable to the BlueDBM system handling everything in flash storage.

The performance of the BlueDBM system is much faster than the 100% off-the-shelf flash system, because how it manages flash storage fits much better with the random access characteristics of this workload. In this experiment, the performance gain from the in-storage accelerator is minimal because hamming distance is such a simple metric.

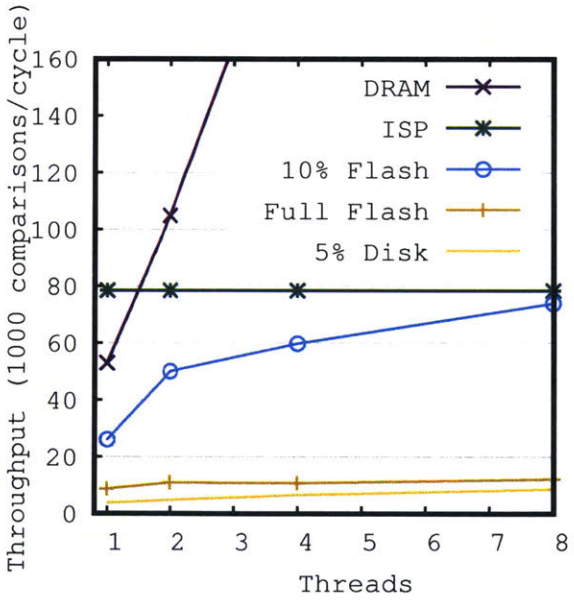


Figure 6-4: DRAM-based system performance drops sharply even when a fraction of requests overflows from memory

6.3.3 Power Consumption Evaluation

We have measured the power consumption to determine the power performance characteristic of our system. The power consumption of a node while running a nearest-neighbor query with images was measured, and the power consumption per amount of work done was calculated by dividing it by the performance numbers obtained from the earlier experiments. The results can be seen in Figure 6-5.

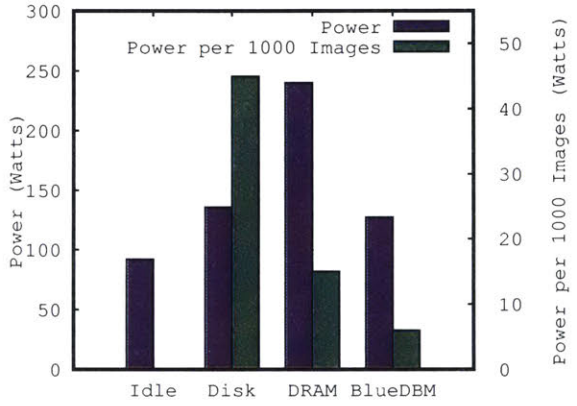


Figure 6-5: Power consumption of various system configurations for nearest-neighbor search

The figure shows that while running everything in DRAM consumes the most amount of power, power consumption per amount of processed data of the DRAM-based system is lower than the disk-based system because the disk-based system is too slow. The BlueDBM-based system, which uses flash with hardware comparators, consumes less power than both disk-and DRAM-based configurations, and also shows much better power per performance numbers.

6.4 Chapter Summary

In this chapter, we presented the design, implementation and evaluation of an approximate nearest-neighbor search of high-dimensional data. In terms of storage performance, this application is coarse-grained random access-intensive and not sensitive to latency, because each comparison was independent of each other. By using the hardware accelerators to saturate flash bandwidth, our BlueDBM-based imple-

mentation was able to achieve performance comparable to a much costlier in-memory software system that can accommodate all data in memory, even for moderately complex distance metrics. Even for simple metrics like hamming distance, our flash-based system could outperform an in-memory system when the available memory capacity is slightly smaller than the size of the dataset, forcing it to swap to storage. The high performance coupled with low power consumption of the accelerated flash system also resulted in very good power-performance numbers. These results demonstrate that an accelerated flash systems can be an attractive solution for applications with coarse-grained random access and moderate to high computation overhead.

Chapter 7

Terabyte Sort on FPGA-Accelerated Flash Storage

Sorting is one of the most studied problems in computer science, and is crucial in many applications. It is used to organize data for *fast searches*, and used for tasks such as *duplicate detection and removal.* For performance, a DBMS sometimes sorts the two tables of interest before *joining* them, or sorts a table by its keys to create a *clustered index.* Large scale sorting is also a key function in the *MapReduce* paradigm, where the keys emitted from mappers must be sorted before being fed into reducers.

Because sorting is usually used as a component of a larger system, the resource budget allocated for sorting is often limited. However, the computational overhead and memory requirement of fast sorting are ever increasing due to the increasing size of the datasets of interest. As a result, sorting large amounts of data can easily become a performance bottleneck.

In this chapter, we present a system architecture for sorting multi-terabytes of data using flash storage connected to a collection of FPGA-based sorting accelerators that perform large-scale merge-sort in storage. The accelerators include highly efficient sorting networks and merge trees that use bitonic sorting to emit multiple sorted values every cycle. The system implements the four types of sorting accelerators described in Table 7.1, each optimized for a memory fabric available on the system.

We show that by appropriate use of accelerators we can remove all the computation

Type	Description
Tuple Sorter	Sorts an N-tuple in an on-chip register using a sorting network
Page Sorter	Sorts an 8KB (a flash page) chunk of sorted N-tuples in an on-chip block RAM
Super-Page Sorter	Sorts sixteen 8K-32MB sorted chunks in DRAM
Storage-to-Storage Sorter	Sorts sixteen 512MB or larger sorted chunks in flash

Table 7.1: Four types of sorting accelerators optimized for a memory fabric

bottlenecks so that the end-to-end sorting performance is limited only by the sequential bandwidth of DRAM and flash storage. As a result, a projected performance of a single-node implementation with two flash cards is able to achieve performance comparable to the advertised performance of a 21-node MapR cluster [8], at a fraction of power consumption.

In fact, *the most notable characteristic of this system is power efficiency.* The power efficiency of a single node prototype implementation is comparable to the current champion of the TeraSort/JouleSort [149] benchmark. This is especially impressive considering: (1) its power performance can be improved further by simply installing another flash storage device, and (2) almost 100 W of power is consumed by the host server, which is not doing much useful work. We also present a system design with a low-power embedded processor and two storage devices, which has more than double the power-performance of the JouleSort champion. Although due to our system's custom design we do not satisfy the JouleSort criterion, it is a good reference point since our system can be constructed entirely using available hardware.

This chapter is organized as follows: Section 7.1 introduces some background and related research on fast sorting of large data. Section 7.2 describes the internal architecture of the sorting accelerator. Section 7.3 provides performance and power consumption evaluations, and we conclude and summarize in Section 7.4.

7.1 Background

7.1.1 Large Scale Sorting

To handle the complexity of the sorting job on large modern datasets, a general-purpose distributed software environment, such as Apache Spark [188] and Hadoop [174], is typically deployed to manage cluster resources and schedule parallel executions. Works in this direction [80, 171, 182, 61] show that a 100 TB of data can be sorted in hundreds of seconds on cluster of general-purpose servers. TencentSort [80] holds the world record of sorting 100 TB of data in 98.8 seconds, which runs on top of 512 OpenPower computational nodes, connected via 100 Gbps Ethernet.

High computation requirement of sorting algorithms has also inspired research into using novel parallel features from off-the-shelf hardware to speed up sorting tasks. Parallel sorting algorithms, such as merge-sort, can be multi-threaded on multicore processors by processing disjoint data segments in parallel [134]. Furthermore, modern processors provide Single Instruction Multiple Data (SIMD) features to more aggressively exploit data-level parallelism in sorting [73, 74]. There is also research to accelerate sorting with larger-scale SIMD appliances such as GPGPUs [155, 60]. SIMD-enabled processors can deliver strong sorting performance on small-scale data, but such solutions are generally power-hungry.

Hardware accelerators, such as FPGAs [161, 55, 127, 118, 92, 27] and ASICs [187, 168, 56], are also under active investigation to provide power-efficient solutions to sorting. Hardware sorting accelerators can be categorized into sequential sorters and parallel sorters. Sequential sorters [187, 92] are implementations of single-threaded sorting algorithms and can produce 1 number/cycle. On the other hand, hardware parallel sorters [161, 127, 27] often implement Bitonic or odd-even sorting networks [18] and can sort more than two inputs simultaneously. Hardware parallel sorters have better performance than sequential sorters, but they need larger area and more I/O pins [56]. With larger capacity FPGAs such as Xilinx UltraScale+ and Altera Stratix 10, hardware parallel sorters are more popular as the implementation choice for sorting accelerators [161].

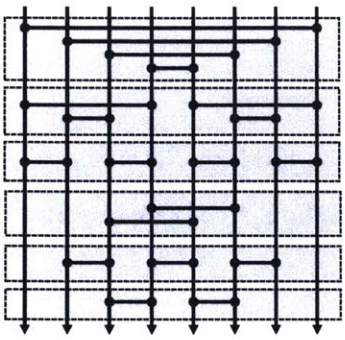


Figure 7-1: A known optimal 8-way sorting network

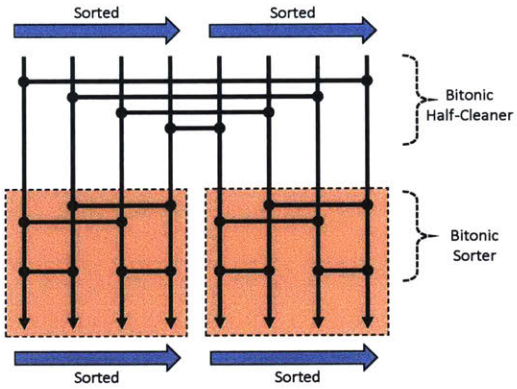


Figure 7-2: Bitonic sorting network

7.1.2 Sorting Network Overview

Sorting networks are computation units that repeatedly perform a sequence of compare and swap operations between pairs of values using comparators. Values are entered into the network in parallel over wires, and when a pair of values meets a comparator, they are compared and swapped, so that each wire now contains $min(x, y)$ and $max(x, y)$ respectively. A sorting network can be made to completely sort a sequence of values in correct order with well-placed comparators. While it is possible to generate a sorting network that sorts the input values of any given length, there is a large body of work to construct *optimal* sorting networks for relatively small input sizes. Such optimal sorting networks have minimal depth, or minimal number of comparators in the network. Optimal sorting networks for the first sixteen input sizes are listed in Knuth's The Art of Computer Programming [91]. Figure 7-1 is the known optimal sorting network for input size of 8, and consists of six stages of comparators.

7.1.3 Bitonic Sorting Network

Sorting networks can take advantage of bitonic sequence characteristics, in which values are either monotonically increasing and then monotonically decreasing, or monotonically decreasing and then monotonically increasing. A bitonic sequence can be

entered into a class of sorting network called *Bitonic half cleaner*, and the output sequence will be separated into two equal-length bitonic sequences, where all values of the upper half will be larger or equal to all values in the lower half. A bitonic half cleaner is a sorting network of depth one. This means the separation of upper and lower halves can be done in a single cycle, in a hardware implementation.

A bitonic half cleaner can be used to merge two sorted sequences efficiently. Given two sequences a and b that are already sorted internally, the concatenation of a and the inverse of b is a bitonic sequence. Therefore, the bitonic half cleaner can be applied to separate the higher values and lower values. Increasingly smaller half cleaners can be applied recursively to merge the two sequences as shown in Figure 7-2. It can be seen that it requires much fewer comparators and less depth compared to the network in Figure 7-1. An important characteristic of this network is that because the upper half and lower half can be separated in a single cycle, a merge-sorter constructed using this unit can merge two sequences organized into units of N values, and emit N sorted values at every cycle. Once the upper and lower values are separated, the merged upper values can be internally sorted in a pipelined fashion. Our sorter will take advantage of this feature extensively.

7.1.4 Sort Benchmark

The Sort Benchmark, colloquially called TeraSort, is a set of benchmarks that measures the capability to sort a large amount of records under a variety of conditions [62]. Initially the main benchmark of interest was the TeraByte Sort, which measures the time to sort 1 TB (10^{12} bytes) of data. Many more benchmarks have been added since to reflect the modern computation environment. One such benchmark of interest to us is JouleSort, which measures the amount of energy required to sort a certain amount of data. Each of the benchmarks in the set comes in two categories: Indy (Formula 1), where records are fixed size (100-byte records with 10-byte keys), and Daytona (Stock Car), where the sort code must be general-purpose.

7.2 Architecture of the Merge-Sort Accelerator

Sorting is done in two phases: in-memory sort and external sort. In the first step of the sorting process which is shown in Figure 7-3, data to be sorted is loaded either from a file in flash or from an input stream, onto DRAM in 512 MB chunks. The loaded chunk is sorted and written to flash. 512 MB is half the size of the on-board DRAM capacity. Three different types of sorters are involved in sorting 512 MB chunks in memory: the Tuple sorter, Page sorter and the Super-Page sorter. Once data in flash is organized into 512 MB sorted blocks, the sorted blocks are iteratively merge-sorted by the Storage-to-Storage Sorter until the entire data is sorted, as shown in Figure 7-4.

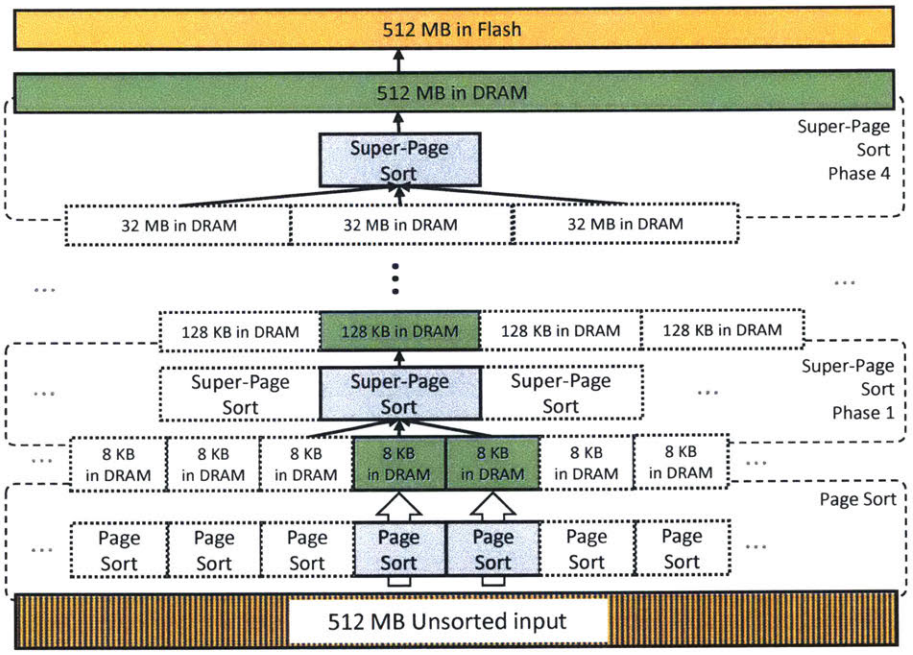


Figure 7-3: Data is first sorted into chunks that can fit in the DRAM buffer and stored back in flash

7.2.1 Tuple Sorter

Our sorting system stores data as packed tuples, which are aligned to the width of the datapath. For example, our implementation has a datapath of 256 bits, and stores data organized into N-tuples that can be packed into 256-bits. 256 bits can

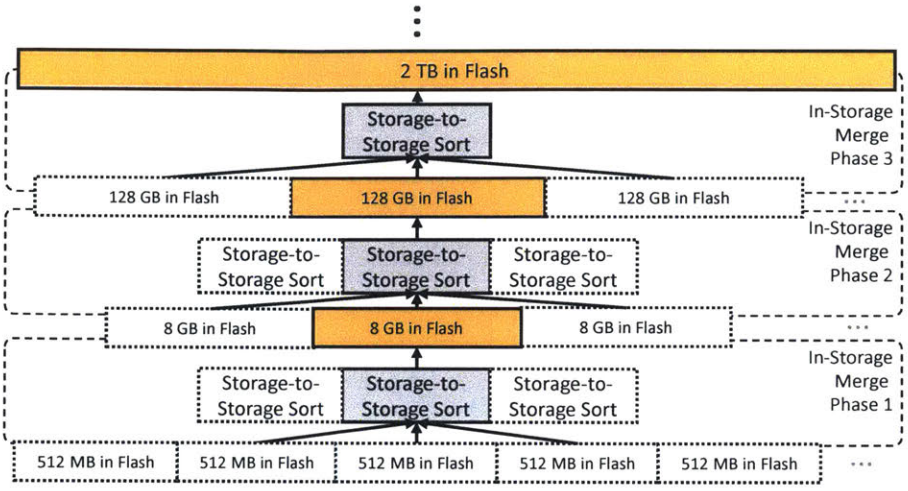


Figure 7-4: Large sorted chunks are merge-sorted directly from storage to storage

fit a 4-tuple of 64 bit values for *long long* values, or a 2-tuple of 128 bit key-pointer pair values for the TeraSort benchmark. Because the size of the tuple is relatively small, an N-tuple can be sorted efficiently using a sorting network. The tuple sorter is located on the datapath of flash reads, so that tuples stored in flash can be sorted as data is read. Because the parallel pipelined sorting network can sort data at wire speeds, only one tuple sorter instance is required. Our sorting system provides a library of known optimal sorting networks that can be selected at compile time.

7.2.2 Merger Sub-Component

All subsequent sorters, including the Page sorter, Super-Page sorter, and the Storage-to-Storage sorter, use one or more instances of the merger module at its core. The merger module takes as input the length of the two sequences to be merged, and the two sequences, each organized into a stream of internally sorted N-tuples. It outputs the merged sequence, also organized into a stream of N-tuples. The merger is capable of emitting a merged N-tuple every cycle, meaning that a certain amount of data can be sorted in a deterministic amount of time, regardless of the size of the values. The internal structure of the merger can be seen in Figure 7-5.

At the beginning of execution, the merger is given the size of sequences to merge. When the first pair of N-tuples enters the merger, they are pushed through the bitonic

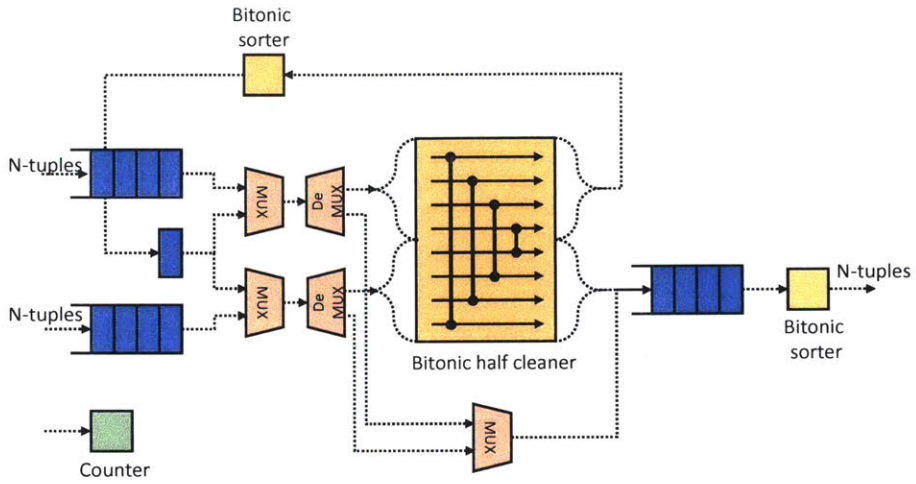


Figure 7-5: Merger internal architecture

half cleaner. When sorting in the ascending order, the values in the lower half of the result form the first N-tuples of the merged sequence. This N-tuple is sorted internally by a bitonic sorter before being output from the merger. The upper half of the results is also sorted by a bitonic sorter, and then stored in a register. All subsequent half cleaner operations are between the value in this register, and one of the input FIFOs. The choice FIFOs that will be used depends on which of the two N-tuples processed in the previous cycle had the larger value. If the N-tuple in the register had the largest value, there are still values in the FIFO that are smaller than the register value. If the n-tuple from the FIFO had the largest value, there may be values in the other FIFO that are smaller than the register value. Once one of the two input FIFOs are empty, the value in the register and in the other FIFO are flushed out without further comparisons.

7.2.3 Page Sorter

The page sorter takes as input a fixed length list of values organized into internally sorted N-tuples, and emits a completely sorted list of the same length. A page-granularity sorter is required because fine-grained random access performance of DRAM drops sharply below page granularity. It makes sense to load page-granularity chunks into on-chip memory and sort them completely. The tuples are sorted by a

sorting network before they are entered into the page sorter.

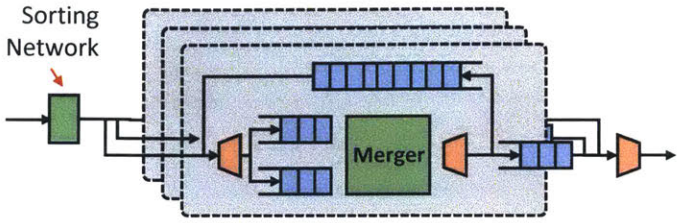


Figure 7-6: Internal structure of a page sorter

The page sorter is used to sort data into page-sized sorted chunks as it is being initially read from flash, or an input stream, at wire-speed. It works by first pushing all N-tuples into one of two FIFOs, merging them into increasingly large chunks of sorted sequences until the whole list is sorted. The internal architecture of a page sorter can be seen in Figure 7-6. Since the page sorter requires multiple passes over the data to sort it completely, multiple instances of page sorters are required to keep up with the bandwidth of the input.

7.2.4 Super-Page Sorter

Once data exists on DRAM as sorted blocks, they are merged into larger chunks with the Super-Page sorter. The Super-Page sorter is composed of two components; An 8-leaf merge tree and a page-granularity DRAM FIFO loader/storer. An 8-leaf merge tree is composed of a tree of the two-way merger described above, and takes 16 streams N-tuples as input and emits a sorted stream of N-tuples. The pairs of stream length information given to the first layer of mergers is added and given to the upper level mergers as input. The internal architecture of an example merge tree with 4-leaf nodes can be seen in Figure 7-7.

The reason such a high fan-out merge sorter is used is to reduce the number of passes the merger has to make to get a fully sorted sequence. Sorting 16 values takes 4 passes with a binary merger, but a single pass with an 8-leaf merge tree. An even larger fan-out will be beneficial to performance, if the on-chip resources of the FPGA allow it. The Super-Page sorter makes multiple passes over data stored in DRAM

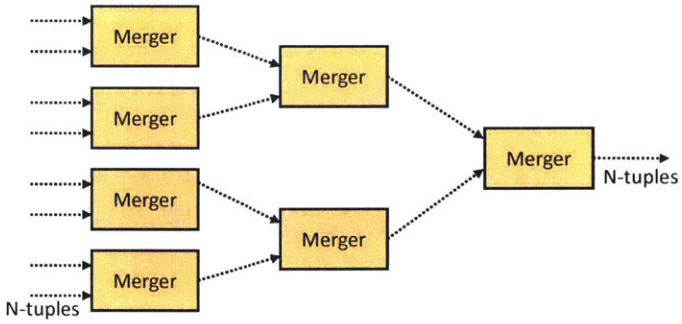


Figure 7-7: A 4-leaf merge tree with 8 N-tuple inputs

until the size of the sorted block becomes half of the available DRAM capacity, at which point it is written to flash.

The DRAM FIFO loader loads page-granularity blocks of data from DRAM and enqueues them into one of the multiple output FIFOs that the merger reads from. It takes as input a stream of DRAM addresses to read from, a stream of DRAM addresses to write to, number of pages to read, and a destination FIFO index. It returns an acknowledgement with the destination FIFO index whenever a read request is fulfilled. The loader initiates a DRAM page read whenever there is enough space on the destination FIFO for a page read. DRAM can be kept completely busy by making sure that one or more requests are always in flight for each FIFO.

The reason a page-granularity loader is required is because DRAM performs relatively poorly with fine-grained random reads. DRAM is organized into banks of a few KBs in size, and there is some overhead whenever a new bank has to be opened. The read performance of our 1 GB SODIMM DRAM card with random 8 KB page-granularity reads was about 10 GB/s, while random 64-byte cache granularity reads performed at about 1 GB/s.

More than one instance of the in-memory merger is usually required to keep up with the maximum bandwidth of DRAM during intermediate merge phases. At the last merge phase, all data in DRAM are collected into a single sorted sequence, and this needs to be done by a single merger. But because the flash write bandwidth is the limiting factor at this stage, one merger is more than enough.

7.2.5 Storage-to-Storage Sorter

Once data is organized into large sorted blocks on flash, they are merged into larger blocks using the Storage-to-Storage sorter. The Storage-to-Storage sorter actually uses the same infrastructure used by the Super-Page sorter, since the Super-Page Sorter is no longer required to be active during the Storage-to-Storage phase. In this phase, the DRAM is used as a prefetch buffer for flash storage. Commands for reading flash pages into DRAM are pipelined with the commands for reading the same pages from DRAM to the merger. The merged pages are also buffered in DRAM, and written back to flash.

7.3 Performance Evaluation

7.3.1 Wire-Speed Assurance of Accelerator Components

All accelerator components perform at wire-speed, meaning computation is never the bottleneck, resulting in the maximum use of the storage and memory bandwidth.

We demonstrated that with 5 page sorters and two 16-way sort mergers, we are able to saturate all system components on a BlueDBM storage device, including the DRAM and flash storage bandwidth. This completely removes the computation bottleneck from sorting, allowing a system to make the best use of its storage and memory bandwidth.

Merger Sub-Component The sort merger was run at a clock frequency of 125 MHz, using a data path of 256 bits. We measured the performance of the merger using values of 64 bits and 128 bits in size. At a data path of 256 bits, the merger takes as input 4-tuples and 2-tuples, respectively. The merger invariably emits one N-tuple at every cycle regardless of data distribution. At 125 MHz, it merges data at 4 GB/s, regardless of value size.

Page Sorter The page sorter sorts 8 KB blocks of data. Since all data is organized into N-tuples with total size of 256 bits or less, each block consists of 256 such tuples.

A two-way merger must make 8 passes over the data to result in a completely sorted block. As a result, each page sorter is capable of producing sorted blocks of data at 0.5 GB/s. In order for the page sorter to sort data as it is read from flash, our system required 5 page sorters to completely saturate the flash bandwidth of a BlueDBM storage device, which is 2.4 GB/s. Depending on the input source of data to sort, the number of page sorters may need to change.

Super-Page and Storage-to-Storage Sorters A 16-way sort merger is capable of emitting a sorted tuple at every cycle, resulting in a 4 GB/s bandwidth per merger. There are three different situations for the 16-way merge-sorter to be used in: Sorting from DRAM to DRAM, DRAM to flash, and flash to flash. The maximum available bandwidth in each situation is 5 GB/s, 2 GB/s and 1 GB/s, respectively. The system must have two 16-way sort mergers in order to saturate the DRAM to DRAM scenario, but only one is required to be active for all other situations.

Table 7.2 describes the sorting phases and the required components to saturate the bandwidth of the medium. For example, during the DRAM-DRAM phase, the 10 GB/s of bandwidth needs to be shared between read and writes, which leaves 5 GB/s each for reads and writes. During the DRAM-flash phase, the DRAM is capable of reading at 10 GB/s, but the flash is only capable of writing 2 GB/s, turning into the bottleneck.

Sort Phase	Bandwidth (GB/s)	Accelerator	Required Instances
Flash Read	2.4	Page Sorter	5
In-memory	5	Merge Tree	2
Flash Write	2	Merge Tree	1
In-storage	1	Merge Tree	1

Table 7.2: Required components for sort phases

7.3.2 Software Performance Comparison

We compared the performance of the merge-sort accelerator against `std::sort`. The software implementation was compiled with -O3 optimizations and run on a Xeon E5-2690. Figure 7-8 shows the elapsed time while sorting 32 GB of 4 byte integers using multiple threads, as well as hardware external and in-memory sorters. We were unable to run larger datasets in-memory on our server machine. Table 7.3 describes the various system configurations we used to evaluate its performance.

Name	Description
SW 1T	Single-thread software
SW 4T	Four-thread software
SW 8T	Eight-thread software
FPGA+F	Accelerator with flash storage and 1 GB DRAM
FPGA+D	Accelerator with enough DRAM

Table 7.3: Systems configurations to evaluate sorting performance

It can be seen from Figure 7-8 that a single instance of the merge-sorter with only 1 GB of DRAM can perform on par with a 4-thread software implementation. With enough DRAM, a single merge-sorter instance can outperform a 8-thread software implementation. Certainly if more memory bandwidth is available, more accelerator instances can be instantiated to achieve even higher performance.

7.3.3 TeraSort Performance

We focused on the Indy category of the TeraSort benchmark, which sorts fixed-sized key-value pairs, to compare our system against existing ones. The Indy category requires sorting of fixed size elements of 100 bytes in size, with 10 byte keys. We generated 1 TB of such key-value pairs, and stored the key and value data separately. The keys were augmented with a pointer to their corresponding values. The flash controller is configured such that when reading a certain key, it automatically reads the corresponding value data. When reading, the on-board DRAM buffer is used as page caches. As a result, the actual dataset to be sorted was a list of 16-byte key-pointer pairs.

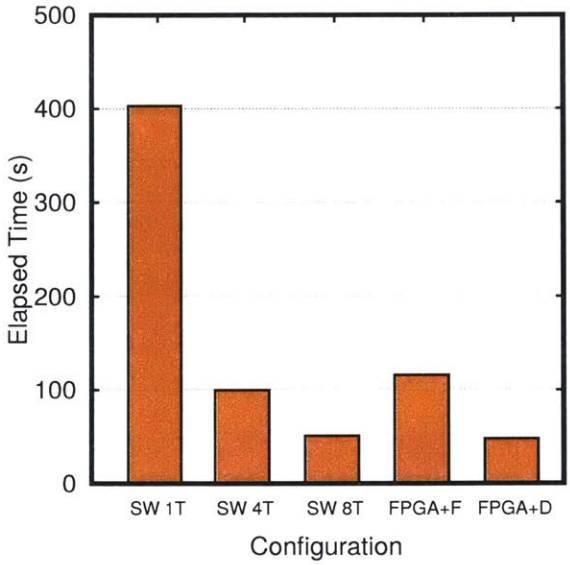


Figure 7-8: Elapsed time for sorting 32 GB of 4 byte integers

It takes our system 700 seconds to completely sort 10^{10} 16-byte key-pointer pairs, or 150 GB of data. However, simply sorting the keys would result in fine-grained random accesses when reading the values, killing performance. To mitigate this issue, every time a 32 MB chunk of the keys is sorted, its corresponding values are read, and written back in a sorted order. Storing values in a partially sorted order results in optimal usage of the page caches, even with random accesses into values that are generated during sequential accesses into keys.

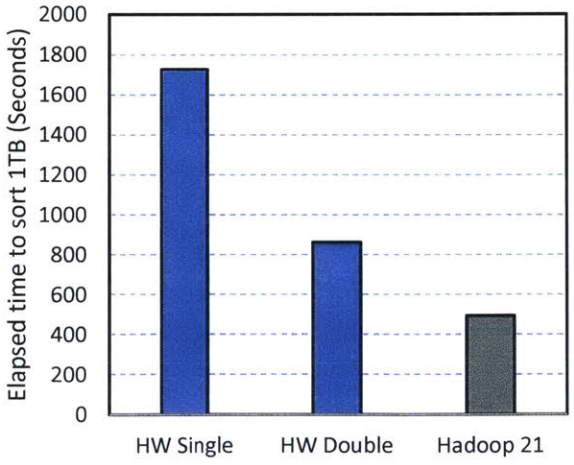


Figure 7-9: Single node performance becomes comparable to a 21-node Hadoop cluster with two accelerated storage devices

Published performance numbers on TeraSort from MapR reports 494 seconds to completely sort 1 TB of data, or 10^{10} keys, on a 21-node cluster, where each node was equipped with 32 cores, 128 GB of RAM and 11 HDDs [8]. Considering the difference in machine capacity, our performance numbers are satisfactory. The single-node performance of our implementation can be improved simply by adding another accelerated storage device. The comparison between accelerated systems with one or two storage devices and the 21-node Hadoop system be seen in Figure 7-9. With a single storage device, our system performs at more than half the performance of a 21-node MapR cluster.

It should be noted that the performance comparison against the MapR system is intended to provide a reference for capability and performance estimate, not to compare the merits of the two architectures. Single node systems such as ours have different constraints compared to cluster systems.

7.3.4 JouleSort Performance

Thanks to the low utilization of the host CPU and the high power efficiency of the FPGA accelerator, the end-to-end power consumption of the system is very low. Not only can a single node system can perform on par with cutting edge cluster systems, but also with much less resources. Thanks to offloading computation to the FPGA accelerator, the host CPU is doing very little work. The host server can conceivably be swapped out with a low power embedded processor without performance loss.

We measured the power consumption of the overall system using a power monitor. The overall system consumed approximately 140 W of power, of which a single accelerated storage device is responsible for about 40 W. It should be noted that our storage implementation is a prototype based on a very conservative power estimation. Production systems will have much lower power consumption.

Figure 7-10 compares the power performance numbers between a fully software implementation of the system, our prototype system with single or double accelerated storage devices, and projected systems with a low-power embedded processor projected to consume 40 W of power. TeraSort benchmark data is packed into 128

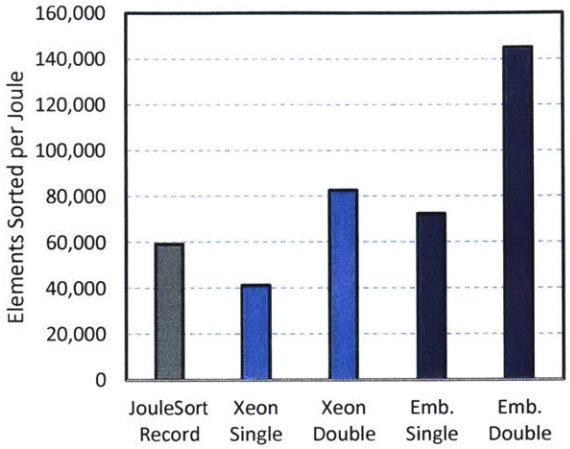


Figure 7-10: Unoptimized prototype achieves power-performance comparable to current JouleSort record. Better implementation is expected to achieve over 2 times the power-performance.

bits of key-pointer pairs. The software numbers are from NTOSort [50], which holds the current record for JouleSort. Not only does our prototype with two flash cards outperform the current record holder, but also a projected system with more realistic components for deployment outperforms the current record holder by over two times.

Although our custom hardware violates the constraints for the JouleSort benchmark, it is a good reference point to compare against. There are also still many improvements that we can make to achieve better power-performance. For example, adding more flash cards will improve power efficiency. For comparison, the NTOSort 1 TB implementation had 16 SSDs installed. Another improvement may be building a more power-optimized accelerated flash storage, as our prototype minFlash board was not designed with power efficiency in mind.

7.4 Chapter Summary

In this chapter, we presented the design and implementation of a low-power high-performance system for sorting terabyte-scale data. Our design used a hierarchy of storage devices and a library of FPGA-based in-storage sorting accelerators to exceed the performance of much larger clusters with a single, much cheaper node. Thanks to the power efficiency of FPGA and flash storage, our system was also able to double

the power efficiency of the current JouleSort record holder.

This sorting system was designed to be one of the key components of a larger in-storage accelerator platform for low-power high-performance graph analytics. More specifically, it is one of the integral components of the terabyte-scale graph analytics system GraFBoost, which will be introduced in Chapter 9.

Chapter 8

Wire-Speed Accelerator for Database Aggregation Operations

Aggregation operations are some of the most fundamental functions of a database system or an analytics platform. They are often implemented as part of GROUPBY or JOIN database operations, where all rows that share a key are collapsed by applying the aggregation function to each row value. User-defined aggregation operations also play an integral part in many analytics platforms. For example, in MapReduce, the user-defined reduce function is effectively an aggregator operator working on a stream of sorted key-value pairs emitted by the shuffle operator provided by the platform. A typical aggregator operator is parameterized with an aggregator function, and takes as input a stream of key-value pairs, in which duplicate keys exist. The aggregator emits a stream of key-value pairs, in which each key is unique, and its value is the collapsed result of applying the aggregator function on all values that were paired with the same key in the input stream. Common aggregation functions in a database include sum, average, count, maximum and minimum, and they are used to collapse the values of multiple rows into a single meaningful value. In this paper, we focus on a scenario in which the input stream of key-value pairs are sorted according to the key, and the aggregator function is associative in nature. This assumption is true for a significant amount of database and analytics workloads, including all database aggregation functions in many databases and analytics applications such as PageRank

on MapReduce.

High performance hardware implementation of aggregation operators is not straightforward, especially when an aggregator function has multi-cycle latency. An important example of a multi-cycle aggregator function is floating point multiplication, which is the aggregator function for PageRank. Using a pipelined Xilinx floating point core, it takes 7 cycles to complete. Due to dependencies, a naïve implementation with a single-aggregator function instance will wait idly for the aggregator function to return, every time a pair of matched keys is discovered.

One way to hide this latency is to have multiple aggregators organized into a tree, and perform aggregations on units of N elements in parallel. However, the number of elements that share a key may be vastly larger than the number of parallel processing elements, which will result in an incompletely aggregated result, as can be seen in Figure 8-1. Simply adding more aggregation units to the tree is not an efficient solution, because the number of elements that share a key can have huge variations, and for some keys can exceed millions of elements depending on the application. As it is not tractable to provision an aggregator tree of an arbitrary size, at some point it must suffer the latency penalty to handle remaining duplicates one-by-one. Furthermore, a tree structure has an inherent performance limitation of emitting up to one element per cycle. Another way to hide latency is to partition the input data into multiple contiguous chunks, apply aggregation on individual chunks, and then merge them together. While this approach can achieve high performance, it requires valuable on-chip memory to store large enough chunks to hide the aggregator function latency.

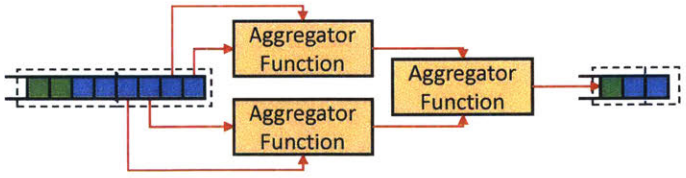


Figure 8-1: Simple parallel aggregator tree may produce incompletely aggregated results

In this chapter, we present an accelerator architecture for wire-speed aggregation operations on a wide datapath, in a bump-in-the-wire fashion. Our accelerator can

maintain wire-speed even when (1) aggregation functions have multiple-cycle latency, and (2) the input stream is wide, i.e., multiple elements arrive every cycle. This is achieved by first striping the wide input across an array of *single-aggregators*, each of which can perform aggregations on a pair of elements every cycle. The resulting streams of aggregated elements are merged together using a hierarchy of *multi-aggregators*, which use sorting network-based high-throughput merge-sorters and a sorting-network-like aggregator network that operate on the duplicate keys that may exist between the streams. All components in the accelerator are pipelined, so that high performance can be achieved without requiring significant on-chip buffers. To our knowledge, our architecture is the only example of a hardware accelerator that can perform wire-speed aggregation on a wide datapath in a bump-in-the-wire fashion, using multi-cycle aggregator functions.

We evaluated the performance of our accelerator using configurations including key-value pairs with 4-byte integer keys and single precision floating point values, and using a pipelined Xilinx floating point multiplier with 7 cycle latency as the aggregator function. One instance of our accelerator could ingest a 4-tuple at every cycle at 125 MHz, achieving 4 GB/s bandwidth regardless of key distribution. Our single-thread software implementation of the same functionality was able to achieve between 0.6 GB/s to 2 GB/s, depending on key distribution. Even assuming linear performance scaling with more threads for the software aggregator, a single instance of our accelerator is capable of outperforming an unlikely software implementation dedicating 6 CPU threads solely for aggregation operations in a database or analytics platform.

Unlike other applications introduced in this thesis, this application does not involve flash storage by default, and is designed as a standalone accelerator that can be incorporated into other systems. This accelerator plays an integral part in the terabyte-scale graph analytics system introduced in Chapter 9.

This chapter is organized as follows: Section 8.1 introduces background and existing research related to the wire-speed aggregator accelerator. Section 8.2 describes the internal architecture of the aggregator accelerator. Section 8.3 demonstrates the

performance of our accelerator. We conclude with a summary in Section 8.4.

8.1 Aggregator Accelerator Background

Database acceleration using reconfigurable hardware accelerators is an actively researched topic, and is one of the most promising directions for continued single-node performance scaling within a power budget. There have been active research in FPGA-based query acceleration engines, which handle a subset or all of the four major database queries: projection, restriction, aggregation and join, as well as some data management functions like row decompression [162].

There has been various designs for such query accelerators. Glacier [126] has explored taking a query and compiling it into a dedicated accelerator. Some works have focused on using dynamic partial reconfiguration functionalities to adapt to the query being accelerated [47, 46, 21]. Systems like Ibex and others have explored database accelerators on the storage device [178, 90]. IBM's Netezza [160] is a commercial product that allows query offloading to a near-storage accelerator. Other systems have implemented accelerators on FPGA fabric with shared memory with the CPU [139, 159].

Additionally, there has been active attempts to incorporate FPGAs into analytics platforms such as Spark or Hadoop, in order to accelerate applications including machine learning and bioinformatics [66, 33, 106, 129, 37].

While efficient implementations of aggregation accelerators have been explored in the past [135, 9], all of hardware aggregator designs we are aware of either (1) cannot process more than one element per cycle, or (2) cannot handle aggregator functions that have multiple cycle latencies. The latter is especially important for modern applications of interest such as machine learning, which requires floating point operations. The Xilinx floating point core requires 7 cycles to perform a single-precision floating point multiplication [181]. This latency can cause significant performance degradation if not handled efficiently.

One of the key components of our aggregation accelerator is a hardware parallel

merge-sorter, which can merge two sorted streams into a new sorted stream, at a rate of multiple elements per cycle. Our accelerator uses the merger sub-component already introduced in Section 7.2.2 for this purpose.

8.2 Accelerator Architecture

Figure 8-2 shows the overall architecture of the aggregator accelerator that processes two elements per cycle, where each unique key is depicted using a different color. The accelerator takes as input a stream of key-value pairs organized into tuples, so that two elements are ingested every cycle. It emits a stream of aggregated key-value pairs also organized into a stream of tuples. The accelerator is composed of two pipelined components: An array of one-element-per-cycle aggregators, or single-aggregators, and a multi-elements-per-cycle aggregator, or multi-aggregators, that combine the results of all single-aggregators. Each single-aggregator and multi-aggregator is also internally composed of multiple components, which will be described in the following subsections.

Because each single-aggregator has a throughput of up to one element per cycle, the number of single-aggregators required is determined by the number of incoming elements per cycle at the input stream. The input stream is partitioned across each single-aggregator in a round-robin way. In other words, given an N-tuple input at each cycle, element 0 in the tuple is routed to single-aggregator 0, element 1 to aggregator 1, and so on. The interleaved distribution of elements across single-aggregators assures no head-of-line blocking at the multi-aggregator stage, because all single-aggregators are operating on similar points in the original stream.

8.2.1 Single-Aggregator

A single-aggregator takes an input stream at a single element per cycle, and emits a stream of aggregated elements at up to a single element per cycle. It ensures two characteristics: (1) all sets of elements in the input stream sharing the same key have been aggregated so that all keys are unique, and (2) there is no backpressure,

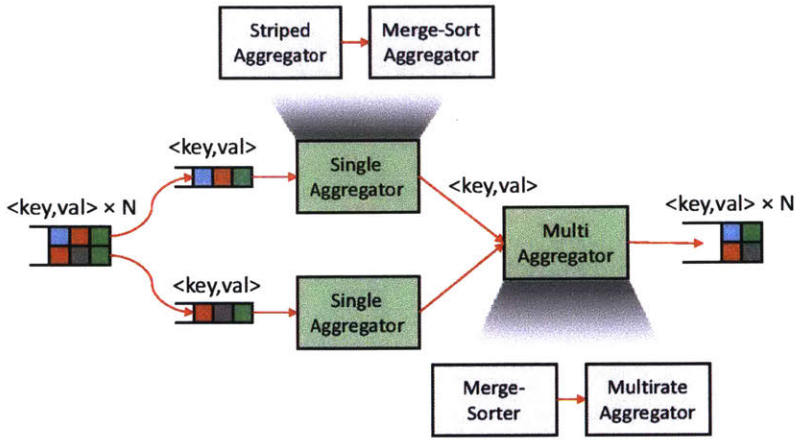


Figure 8-2: Accelerator architecture overview

ensuring that the single-aggregator does not cause performance degradation in the stream, even with aggregator functions with multicycle latency.

Each single-aggregator consists of two pipelined stages, the striped aggregator and the merge-sort aggregator.

8.2.1.1 Striped Aggregator

The architecture of the striped aggregator can be seen in Figure 8-3. The striped aggregator has a parameter *stride*, which is determined by the latency of the aggregator function. For example, a striped aggregator using a simple function like MAX or MIN may have a stride of 1, whereas one that uses a floating point multiplier will have a stride of 7. It ensures that each key in its output has a maximum of *stride* duplicates. A striped aggregator stores the key-value pair elements in the last stride in a FIFO, and compares them sequentially against the elements in the next stride. If the key of an incoming element does not match the key of the element in the same position of the last stride, the element from the FIFO is emitted into the output FIFO, and the new element is pushed into the last stride FIFO. If the keys do match, the matched key is pushed into the last stride FIFO, coupled with a value tagged invalid, and the values from the two matched elements are pushed into the aggregator function, so that the calculation results are ready the next time the same key is compared.

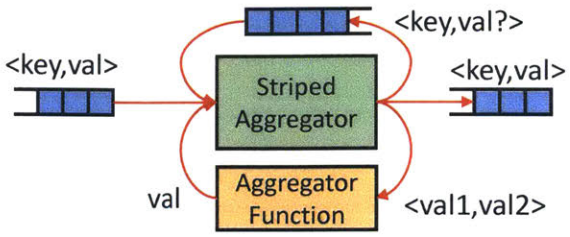


Figure 8-3: A striped aggregator for using an aggregator function of latency 4 ensures each key in the output stream has a maximum of 4 duplicates

8.2.1.2 Merge-Sort Aggregator

The merge-sort aggregator tree takes the output from the striped aggregator, which is ensured to have less than *stride* duplicate per key, and reduces them further to emit a single stream with no duplicates. The architecture of a merge-sort aggregator tree of *stride* 4 can be seen in Figure 8-4. It stripes the input stream into *stride* FIFOS, and then performs hierarchical merge-sort across the striped streams. The internal architecture of a merge-sort aggregator node is shown in Figure 8-5. At each merge-sort aggregator node, whenever the two input elements have the same key, it pushes the two values into the aggregator function, and emits the result along with the shared key. Otherwise, the incoming key-value pairs are emitted one-by-one in order. Because the tree of merge-sorters ensure that all pairs of values within *stride* distance will be compared against each other, the output from the merge-sort aggregator tree is ensured to be completely unique in terms of keys. Also the striped distribution assures that there is no be head-of-line blocking at merge-sorters even with very small FIFOs.

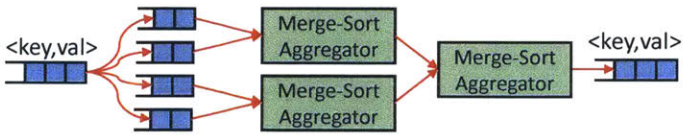


Figure 8-4: Merge-sort aggregator tree for four streams

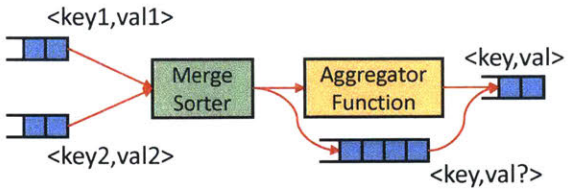


Figure 8-5: Internals of a merge-sort aggregator

8.2.2 Multi-Aggregator

The multi-aggregator stage takes in multiple aggregated streams emitted by the array of single-aggregators and merges them into a single wide, aggregated stream. The merging is done in a hierarchical fashion. For example, when output from four single-aggregators must be merged into a single stream, pairs of streams are first merged and aggregated into two streams of width two. Then, the two streams can be merged and aggregated into a single stream of width four.

Each layer of the multi-aggregator hierarchy consists of 2-to-1 multi-aggregators. The internals of a 2-to-1 multi-aggregator can be seen in Figure 8-6. Each 2-to-1 multi-aggregator consists of two pipelined stages: merge-sort and the multirate aggregator. The multirate aggregator is responsible of making sure the resulting wide stream is fully aggregated, and does not have any duplicate keys. The design of the multirate aggregator is greatly simplified by the uniqueness assurance given by the each of the single-aggregators. As a result, all input given to each 2-to-1 multi-aggregator has an upper bound of two elements with the same key, for each key.

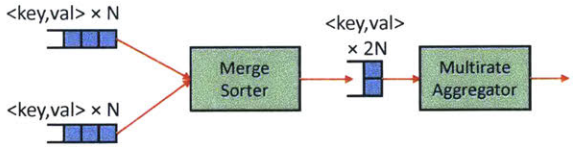


Figure 8-6: A multi-aggregator takes in two aggregated streams of width N and emits an aggregated stream of width N×2

8.2.2.1 Multirate Merge-Sort

Figure 8-7 shows the internal design of a 2-to-1 merge-sorter. The design of such a merge-sorter is described in some of the previous research on multirate merge-sorters

for FPGAs [161]. Our implementation uses a sorting network construct called a *bitonic half-cleaner*, which takes a bitonic sequence and separates the higher values from lower values. Given two sorted tuples, simply reversing the order of one and concatenating them together forms a bitonic sequence, which can be operated on by the bitonic half-cleaner. The lower values from the bitonic half-cleaner, which is also bitonic in nature, can be pushed through a sorting network construct called a bitonic sorter, which sorts a bitonic sequence. The higher values from the bitonic half-cleaner can also be pushed through a sorting network before being buffered on a register. Depending on which of the two input streams had emitted the lowest value in this cycle, the buffered value is compared against the next value from that source stream at the next cycle.

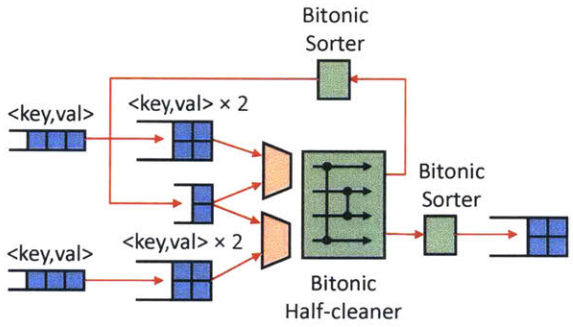


Figure 8-7: Internals of a 2-to-1 merge-sorter

8.2.2.2 Multirate Aggregator

The merged, multirate stream may have duplicate keys that existed in both source streams. However, because the keys in the two source streams are known to be unique, each key in the resulting stream can have a maximum of two instances, making their handling simpler.

Because there are up to two elements with duplicate keys for each key, there are only two possible locations for such pairs: across tuple boundaries between two tuples, or contained within a tuple. Therefore, identifying and aggregating such pairs is done in two pipelined stages: across tuple boundaries and internal to tuples. Figure 8-8 shows the internal architecture of a multirate aggregator dealing with 2-tuples.

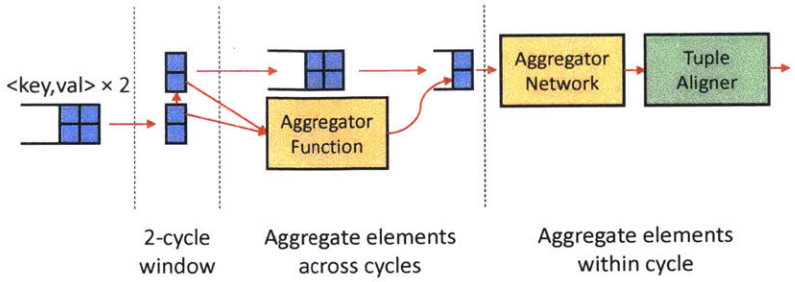


Figure 8-8: A multirate aggregator removes duplicates first across tuple boundaries, and then internally

First, two most recent tuples are buffered, so that the pair of keys across tuple boundaries can be compared. If the keys match, the values are pushed into an aggregator function, and the first element in the later tuple (which is one of the pair of elements) is marked invalid. If the keys do not match, the tuple is simply forwarded.

After this step, it is assured that no duplicate keys exist across tuple boundaries, and only pairs internal to each tuple need to be compared. For example, in a 2-tuple multirate aggregator, aggregation within a tuple can be done simply by comparing the two keys in the tuple and applying the aggregator function in a pipelined fashion, if necessary. For an aggregator handling a larger tuple, aggregation within a tuple needs to be done using a sorting network-like construct to deal with aggregations in a pipelined way, which we call an aggregator network.

8.2.2.3 Aggregator Network

An aggregator network checks for key-value pairs with duplicate keys within a tuple and applies the aggregator function in a pipelined way. It takes as input an N-tuple for key-value pairs every cycle, and emits an N-tuple of key-value pairs every cycle. The elements in the resulting tuple are assured to be fully aggregated, and have no duplicate keys. Furthermore, the "bubbles" created by reducing two elements with duplicate keys are marked invalid and shifted to the end of the tuple, so that all valid values are aligned to the beginning of the resulting tuple.

An example aggregator network for four elements can be seen in Figure 8-9. The network consists of two pipelined primitives: *compare-and-aggregate*, and *compare-*

and-shift. Compare-and-aggregate compares the keys of two elements, and applies the aggregator function if the keys match. The resulting value is stored in the earlier element's value field, and the later element is marked invalid. If the earlier element is invalid, the later element is shifted to the earlier position. Compare-and-shift simply checks if the earlier element is marked invalid, and shifts the later element to the earlier position if the earlier position is invalid. A 2-tuple aggregator network can be constructed with a single compare-and-aggregate element.

An odd-even sorting network for N-tuples with all compare-and-swap units in the upper left triangle changed to compare-and-aggregate, and the compare-and-swap units in the lower right triangle changed to compare-and-shift is a baseline design for an aggregator network of N-tuples. The 4-tuple aggregator network in Figure 8-9 is an example of such a network. The upper-left triangle of compare-and-aggregate units assure that all possible aggregations are performed, and the lower-right triangle of compare-and-swap units assure that all possible bubbles are propagated to the end of the network. As with the case of sorting networks, more optimized networks may be possible.

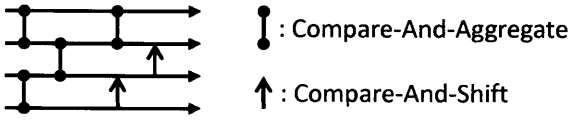


Figure 8-9: A pipelined aggregator network for 4-tuples

8.2.2.4 Tuple Aligner

The resulting stream emitted by the aggregator network may have elements marked invalid, or "bubbles" at the end of each tuple. The tuple aligner packs this stream of elements and emits a stream without bubbles at wire speed, again using a sorting-network-like architecture. The internal architecture of a tuple aligner for a stream of 4-tuple can be seen in Figure 8-10.

The tuple aligner for N-tuples consists of N-1 pipelined stages, and it functions by circular shifting the input tuples so that it will start at the offset at which the previous tuple's valid data ends, and concatenating them by removing the invalid elements. At

the beginning of the pipeline, the tuple aligner remembers the *current offset*, which is the offset of the first invalid element in the last tuple once it has been shifted to the target offset. The *current offset* is the offset the next tuple needs to be shifted to, so after each tuple is pushed into the pipeline, the *current offset* is updated using the number of valid elements in the previous tuple. And the *current offset* value is pushed into the pipeline with the tuple to be shifted, and acts as the number of remaining shifts that must be performed. At every pipeline stage, the tuple is circular shifted by one if the *shifts remaining* value is larger than zero. At the end of the pipeline, only the elements that are valid are pushed into one of the single-element FIFOs, and whenever all single-element FIFOs have values in them, they are dequeued together and emitted. If the tuple is tagged as *last*, the single-element FIFOs are dequeued and emitted even if not all FIFOs have elements in them.

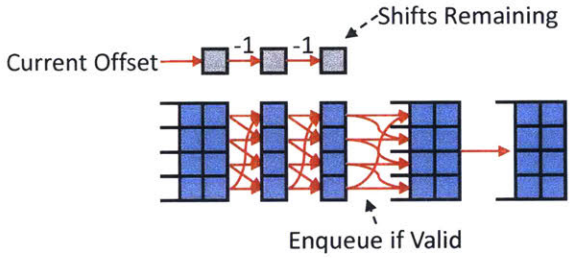


Figure 8-10: Pipelined tuple aligner for 4-tuples

8.3 Performance Evaluation

We have performed two performance evaluations on this accelerator design: (1) wire-speed assurance and (2) comparison against software implementations.

8.3.1 Wire-Speed Assurance

We have tested two different aggregator functions: floating point multiplication and integer maximum. The characteristics of the two aggregator functions can be seen in Table 8.1. The cycle count for the aggregator function includes the cycles for enqueueing and dequeueing values to and from the aggregator function. All examples

were tested with a tuple size of 2 and 4 elements, which adds up to 16-byte and 32-byte data paths, respectively.

	Cycles	Key	Value
Float Mult	8	4-byte int	4-byte float
Maximum	2	4-byte int	4-byte int

Table 8.1: Aggregator functions tested

In all cases, the accelerator was able to saturate the speed of the data bus. We have run all accelerators successfully at 125 MHz, resulting in a throughput of 2 GB/s for all evaluations with 2-element tuples and 4 GB/s for all evaluations with 4-element tuples.

When the aggregator accelerator is used separately to accelerate database or analytics applications, this kind of performance is enough to saturate the bandwidth of many of the data source components. For example, a typical PCIe-attached SSD drive with Gen2 x8 PCIe link has a maximum bandwidth of 4 GB/s. Using 4-element tuples, a single instance of our accelerator is capable of saturating the performance of such a storage device, eliminating the computation bottleneck for storage-backed platforms. Furthermore, it can saturate the host-side link performance of a FPGA accelerator connected to a host machine over PCIe Gen2 x8. In order to exceed this link bottleneck in a database appliance, we must explore emerging architectures such as accelerators in the network interface card (NIC) or the storage device itself. The bump-in-the-wire characteristic of our architecture makes it easy to transparently embed this accelerator to such system components.

8.3.2 Software Performance Comparison

A software aggregator implementation was written in C to compare performance. The C implementation takes as input 4 MB memory blocks packed with 4-byte key and 4-byte value pairs. It cycles through the key-value pairs in the stream of 4 MB chunks while buffering the latest key-value pair in a temporary variable, and performs aggregation whenever a duplicate key is discovered. Whenever a new key differs from

the latest key in the temporary variable, the current buffered key-value pair is written to a pre-allocated memory space, and the new key-value pair is buffered.

We have compared the performance of the hardware aggregator against two software implementations: single thread and projected 4-thread performance. The projected performance of the 4-thread implementation assumes linear speedup compared to the single thread implementation. The software implementation was compiled with -O3 optimizations and run on a Xeon E5-2690. As an aside, it was difficult to achieve higher software performance using SIMD operations such as AVX due to the irregular distribution of keys, as well as the wide key-value pair sizes.

Figure 8-11 compares the performance of the hardware aggregator against the two software implementations, on inputs with differing ratios of duplicate keys. The accelerator performance is constant regardless of key distribution, as performance is completely deterministic. However, the software implementations have performance degradation when there are more duplicates, because executing the aggregation function (floating point multiplication) adds computation overhead. Even the performance of the 4-thread software implementation with ideal linear scaling becomes slower than the hardware implementation when the average number of duplicate keys exceed 8. The software performance seems to level out after 512 duplicates, and demonstrates similar performance even on a dataset with only one key. Considering that the ratio of duplicate keys is generally much higher than eight for most meaningful applications including PageRank, the performance of the hardware accelerator is desirable compared to the projected 4-thread software implementation.

8.4 Chapter Summary

In this chapter, we presented the design and implementation of a wire-speed accelerator for database aggregation, and demonstrated that our accelerator was capable of maintaining wire-speed performance even when the input stream is multirate (multiple elements per cycle via a wide datapath), and the user-defined aggregation function has a multi-cycle latency. Given a stream of key-value pairs sorted by the key, our

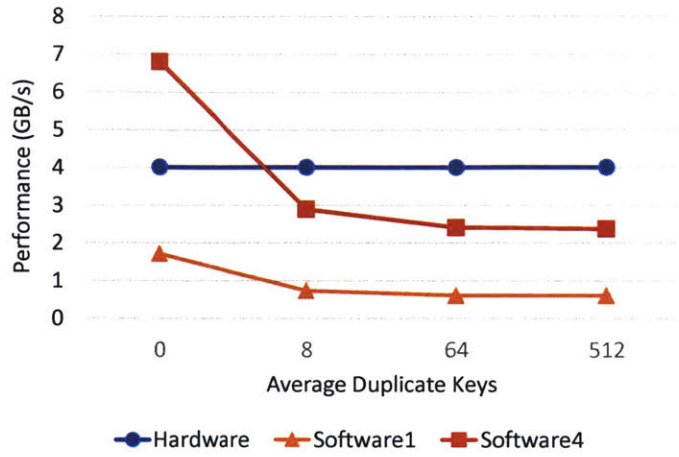


Figure 8-11: Floating point multiplication aggregation performance comparison against software implementations

accelerator is capable of aggregating these elements in a bump-in-the-wire fashion, without using a significant amount of memory.

We have designed this accelerator to function as a component of a larger system, such as a database management system or an analytics platform such as Spark or Hadoop. Due to the bump-in-the-wire nature of our accelerator operation, it can be incorporated transparently into the analytics datapath. And because a single instance of the accelerator is capable of saturating the bandwidth of a 10G Ethernet or a PCIe Gen2 x8 link, which is the link of choice for most PCIe-attached SSDs, it will not become a performance bottleneck in such a system. Furthermore, by freeing 6 or more software threads from the burden of working on aggregation, the rest of the system can focus computation capacity on other important tasks, improving performance and/or reducing cost.

This accelerator also plays an integral part in the GraFBoost terabyte-scale graph analytics system introduced in Chapter 9.

Chapter 9

GraFBoost: Graph Analytics on Secondary Storage

In this chapter, we describe GraFBoost, a system for analytics of multi-terabyte graphs in flash storage. Unlike other high-performance graph analytics platforms which require either all graph data or at least algorithmic vertex data to be completely accommodated in DRAM, GraFBoost only requires that all graph data fit in local flash storage. Thanks to the low cost per capacity of flash storage relative to DRAM, a much cheaper GraFBoost system can handle an order of magnitude larger graphs than an in-memory system of similar cost. Unlike in-memory and semi-external systems, GraFBoost uses a constant amount of memory for all problems, allowing GraFBoost to scale to much larger problems than possible with existing systems while using much less resources on a single-node system.

The key component of GraFBoost is a novel algorithm called sort-reduce and its implementation as a hardware accelerator. Graph analytics is inherently fine-grained random access intensive both for read and write, and is traditionally a bad fit for coarse-grained secondary storage devices. Sort-reduce is an algorithm that efficiently sequentializes a stream of fine-grained random updates, or read-modify-write, operations. Sequentializing accesses using sort-reduce can prevent the thousandfold performance degradation that could be caused by fine-grained updates with random, irregular access patterns. Sort-reduce is constructed in two parts: First, it logs and

sorts the stream of update requests according to the address to update. Second, given an update function that is associative, it also interleaves sorting with in-place update operations whenever possible, drastically reducing the algorithm overhead in the applications we tested. These two parts are called *sort* and *reduce*, respectively. While sort-reduce was initially designed for efficient graph analytics in secondary storage, it can be useful for many other applications that require fine-grained random updates.

We have implemented two graph analytics systems using the sort-reduce algorithm: a purely software implementation, called GraFSoft, as well as an implementation with hardware acceleration, called GraFBoost. Given the same hardware platform, GraFSoft is capable of handling much larger graphs than existing graph analytics systems. And while small graphs that can entirely fit inside the DRAM of a single machine are not the focus of this work, GraFSoft still demonstrates performance comparable to state-of-the-art software systems. An ideal solution will be a graph analytics system that uses sort-reduce when graph sizes are larger than available memory, and uses conventional methods when they are not. GraFBoost uses hardware accelerators to make the best use of memory and storage bandwidth at much lower power consumption. It also uses the high-performance flash interface of BlueDBM to consistently achieve maximum available storage bandwidth.

We compare the performance of GraFSoft and GraFBoost against various state-of-the-art graph analytics software including FlashGraph. GraFSoft and other graph analytics software are run on a 32-thread Xeon server with 128 GB of DRAM. GraFBoost was implemented on BlueDBM using the in-storage hardware accelerator equipped with 1 GB of DRAM. We demonstrate that GraFSoft and GraFBoost both require relatively small amount of host DRAM (8 GB and 2 GB, respectively), but achieve high performance with very large graphs no other system can handle. While the primary interest of GraFBoost is large graphs that cannot fit in memory, GraFBoost systems still rival the performance of the fastest software platforms on graphs small enough for existing platforms to handle. GraFBoost consumes a fraction of power compared to GraFSoft, thanks to hardware acceleration, but performs even faster, due to both hardware acceleration and the high-performance storage interface.

This chapter is organized as follows: Section 9.1 provides background and existing research related to large graph analytics. Section 9.3 introduces the novel sort-reduce algorithm for sequentializing fine-grained random updates. In Section 9.4, we present algorithmically how sort-reduce can be applied to remove all random accesses from the push-style vertex-centric graph analytics. Section 9.5 describes the architecture of GraFBoost, which uses the sort-reduce accelerator implemented on BlueDBM. Section 9.6 describes the prominent parts of the GraFSoft implementation, which is implemented purely in software. Section 9.7 presents the graph analytics benchmarks and the comparison systems we ran them on, and demonstrates the performance and low resource utilization of GraFBoost. Section 9.8 provides a summary of this chapter.

9.1 Background

Extremely large and sparse graphs with irregular structures (billions of vertices and hundreds of billions of edges) arise in many important problems, for example, analyses of social networks [86, 16], and structure of neurons in the brain [104]. Their efficient processing enables everything from optimal resource management in power grids [54] to terrorist network detection [184].

The irregular nature of graphs makes graph analytics fine-grained random access-intensive, and typically requires all data to fit in DRAM [112, 157], as opposed to slower secondary storage. However, as graphs become larger, they quickly become too large to fit in the main memory of a reasonable machine, and must be distributed across a cluster [138]. Large-scale graph analytics are usually done using a distributed graph processing platform following a programming model it provides, so the user does not have to deal with the difficulties of distribution, parallelism and resource management [89, 189, 172, 183].

9.1.1 Models for Graph Analytics

Vertex-Centric Model Many prominent graph analytics platforms, including Pregel [116], Giraph [15] and GraphLab [110], expose a vertex-centric programming model [120] because of its ease for distributed execution. In a vertex-centric model, a graph algorithm is deconstructed so that it can be represented by running a *vertex program* on each of the vertices. A vertex program takes information about the current vertex as input, as well as its neighboring vertices and the edges that connect them. After execution, a vertex program updates the current vertex, and possibly sends messages to neighboring vertices.

Vertex-centric graph analytics systems can be categorized into two groups depending on how vertex programs are scheduled:

- ***Bulk synchronous manner***: Algorithm execution is organized into a series of disjoint iterations, or *supersteps* [116, 153, 193]. In a superstep, the programmer-defined vertex program is executed on all active vertices, and determines a set of active vertices for the next superstep. A vertex program in a superstep can only read from the previous superstep.
- ***Asynchronous manner***: Each vertex program sees the latest value from neighboring vertices, and can execute asynchronously whenever information is available [110, 59].

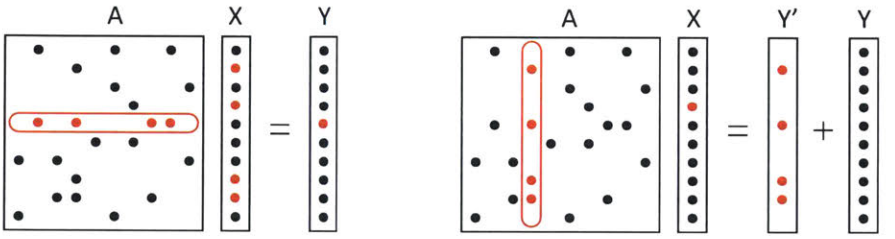
While asynchronous systems are more complex compared to bulk synchronous systems, sometimes they result in faster convergence for many machine learning problems [110].

Vertex-centric systems can also be categorized according to the data available to the vertex program:

- ***Pull-style***: The vertex program reads the values of neighboring vertices and updates its own value. All neighbors for each vertex are read together, so that each vertex's value is updated once, but are read many times across algorithm execution.

- ***Push-style***: The vertex program updates the values of its neighbors. Push-style updates all neighbors of each vertex at once, so that each vertex's value is read only once, but updated as many times as its incoming neighbors.

The difference between the two categories can be seen in Figure 9-1. The two figures represent graph structure, or edge data, represented as an incident matrix. The algorithmic state, or vertex data, is represented as an array of vertex values that is updated through algorithm execution.



(a) Pull-style vertex programming model

(b) Push-style vertex programming model

Figure 9-1: Two categories of the vertex-centric model

Many of the models introduced below, can be thought of as variations of the vertex-centric model.

Gather-Apply-Scatter Model Vertex-centric systems running on a system with multiple cores typically assign each vertex program to a core. However, because many graphs found in nature follow a power-law distribution [52], where few vertices have disproportionately large number of neighbors, simply assigning vertices to cores may result in substantial work imbalance [59]. Systems including PowerGraph [59] and Mosaic [114] partition work for even a single vertex program across many processing units, and merge them later for efficient processing, using a modified vertex-centric model called Gather-Apply-Scatter. In order to obtain correct results after merging partial solutions, these systems require parts of the user-defined program, namely the *Gather* part, to be binary-associative.

Edge-Centric Model Some systems such as X-Stream [151] and GridGraph [194] provide edge-centric programming models, which are aimed at sequentializing accesses into edge data stored in secondary storage, which is usually significantly larger than vertex data. Edge-centric systems have the benefits of doing completely sequential accesses to the edge data, but must process all edges in the graph at every superstep, making them inefficient for algorithms with sparse active lists like breadth-first search. In terms of data access pattern and performed computation, the edge-centric model is very similar to the push-style vertex-centric programming model configured to assume all vertices are active at every iteration.

Subgraph-Centric Model Some systems, like Giraph++ [166] and Blogel [186], expand the vertex-centric model into a subgraph-centric model, in which systems process blocks of subgraphs and messages between them. Subgraph-centric systems attempt to reduce the communication overhead of the vertex-centric programming model by exposing to the programmer information about how the graph is partitioned across a cluster and optimizing messages within a single machine.

Linear Algebraic Model Some recent systems such as Combinatorial BLAS [25], GraphBLAS [87], and Graphulo [71] provide the user a set of linear algebra functions designed to map graph algorithms to linear algebra [88]. These systems use the wide range of existing work on high-performance linear algebra operations to efficiently execute graph algorithms. Linear algebraic systems define a vertex program in terms of *Semirings*, an algebraic structure equipped with two binary operations *addition* and *multiplication*. In the context of graph algorithms, an important characteristic of a semiring is that addition must be commutative and associative. Conceptually, the linear algebraic model can be thought as a subset of the push-style vertex-centric programming model, addition and multiplication can be mapped to a vertex program with commutativity and associativity restrictions.

More General Models While such variations of localized models are versatile enough for many important applications, some applications still require finer-grained

control of execution. Galois [131] is one such framework that has high-performance platform implementations both on shared and distributed memory systems.

9.1.2 Managing Changing Graphs

Graph analytics systems are also characterized by whether they allow dynamic modification of the input graph. Many systems focus on achieving high performance on algorithms that do not modify the input graph [158, 190, 151, 114], while some systems optimize for efficient handling of changing graphs [115, 95].

9.1.3 External Graph Analytics

Single-node in-memory systems aim to optimize performance for data that can fit in the memory of a single machine, by making the best use of shared memory parallel processing [158, 131]. However, such in-memory systems require enough memory to accommodate the entire graph, and fail to operate when this requirement is not met. One way to handle graphs which exceed the memory capacity of a single machine is to distribute it across a cluster [109].

Distributed clusters can accommodate very large graph in their collective memory, but are expensive to operate, and suffer from significant distribution overhead. The distribution overhead for graph analytics is often so large that a cluster of 16 servers are sometimes slower than a single-thread implementation on a laptop [122]. Furthermore, as graphs become larger, it is often not financially viable for an entity to construct a large enough cluster to accommodate multiple terabytes of graphs. An actively researched solution to this issue is using a single-node, or a much smaller cluster of machines by storing and processing the graph in secondary storage. Naively using secondary storage instead of memory results in very low performance, due to performance and access granularity issues introduced in Section 2.1.3. Many external and semi-external systems have attempted to solve these issues.

Semi-external systems achieve high performance with less memory by storing only the vertex data in memory, and optimizing access to the graph data stored in sec-

ondary storage. Vertex data is much smaller than edge data, often by an order of magnitude. Semi-external systems such as FlashGraph [190] can often achieve performance comparable to complete in-memory systems while using much smaller amounts of DRAM. However, such systems are not applicable when graph sizes become so large that even vertex data does not fit in memory.

X-Stream [151] is also a semi-external system that achieves high performance by keeping only vertex data in memory, but it can sometimes maintain performance even when vertex data does not fit in memory. It logs updates to vertex values before applying them, so when available memory capacity is low they can easily divide the vertices and the update log into partitions. This kind of partitioning is not readily applicable to vertex-centric systems such as FlashGraph [190], because reading the value of neighboring vertices for each vertex program execution requires fine-grained random reads. A limitation of X-Stream is that because it follows the edge-centric programming model, it processes all vertices at every superstep, making it slow for algorithms with sparse active vertex sets.

When graph sizes become too large even to store only vertex data in DRAM, external systems like GraphChi [95] become the only choice due to their lower memory requirements. GraphChi re-organizes the algorithm to make data access completely sequential, and thus, makes accesses perfectly suitable for coarse-grained disk access. There is also active research on optimizing storage access for such external graph analytics systems [170]. However, GraphChi does so by introducing additional work, and requires the whole graph data to be read multiple times each iteration. These extra calculations result in low performance on large graphs and make GraphChi uncompetitive with memory-based systems. Some systems including LLAMA [115] and MMap [105] use OS-provided `mmap` capabilities, and trust the OS to make intelligent on-request storage access.

Another way to deal with the large amount of graph data is to use a graph-optimized database, such as neo4j [5], which provides an ACID-compliant transactional interface to a graph database. There have been efforts to use generic RDBM engines for graph analytics instead of specialized systems [53].

9.1.4 Special Hardware for Graph Analytics

As performance scaling of CPUs slows down, many systems have attempted to efficiently use newer computation fabric to accelerate analytics, such as domain-specific architectures for database queries [180, 38, 164, 177], in-memory graph-specific processors such as GraphCREST [2], Graphicionado [67] and other processor architectures optimized for graph processing [140, 11, 113, 78]. GraphGen [133] attempts to demonstrate high performance and power-efficient in-memory graph analytics using FPGAs by automatically compiling an application-specific graph processor and memory system. Coupled with good FPGA hardware, GraphGen was able to outperform handwritten CPU and GPU programs sometimes by over 10x. ForeGraph [41] uses a cluster of FPGAs to accommodate a graph in its collective on-chip memory. Platforms such as GunRock [173] organize graph processing to be effectively parallelized in a GPU.

9.2 Algorithmic Representation of Push-Style Vertex-Centric Graph Analytics

In this research, we focus on optimizing the push-style vertex-centric programming model for flash storage. More specifically, we focus on a slightly restricted model in which the vertex program consists of two programs: `edge_program`, which is executed on each outbound edge of an active vertex, and `vertex_update`, which is used to update the neighbor vertex value using the results of the active vertex. These two programs are analogous to the multiplication and addition operations in the linear algebraic model, and we assume the vertex update program is binary-associative. We also assume that the input graph structure does not change, but this is not a fundamental restriction of our methods, rather a characteristic of our prototype implementation.

A superstep in such a graph analytics model can be expressed algorithmically as shown in Algorithm 1. The programmer provides definitions for algorithm-specific

Algorithm 1 A baseline algorithm for performing a vertex program superstep

```
▷ A_0 is algorithm init condition
newV_i = {}                                  ▷ All values initialized to identity element 0
for each ⟨k_src, v_src⟩ in A_{i-1} do        ▷ Iterate through active vertices
    for each e(k_src, k_dst) in G do         ▷ Iterate through outbound edges of k_src
        ev = edge_program(v_src, e.prop)
        newV_i[k_dst] = vertex_update(ev, newV_i[k_dst])
    end for
end for

for each ⟨k, v⟩ in newV_i do
    x = finalize(v, V[k])                    ▷ Apply remaining element-wise calculation
    if is_active(x, V[k]) then               ▷ Whether k should be active at next iteration
        A_i.append(⟨k, x⟩)
        V[k] = v                             ▷ Update value of k
    end if
end for
```

functions `edge_program`, `vertex_update`, `finalize`, and `is_active`.

The active list A_i is a list of key-value pairs, representing the vertex index and its value of the active vertices for iteration i. A_0 is set to be the initial condition of the graph algorithm, and the algorithm is executed starting from i of 1. V is a map from vertex index to its value from the most recent iteration, and $newV_i$ is a temporary data structure, and is a map from vertex index to its temporary value as it is being calculated during iteration i.

In each *superstep* i, it goes through the current *active vertex* list A_{i-1}, and computes the new temporary vertex values $newV_i$ using the *edge_program* and *vertex_update* functions. Whenever $newV_i[k_{dst}]$ for a previously unvisited k_{dst} is accessed, its value is set to ev, because $newV_i$ is initialized with the identity element 0 in regards to the vertex update function. We then compute the new vertex values by applying the *finalize* function to each element of $newV_i$. The finalize program can use its value from the previous superstep using $V[k]$. In many algorithms including breadth-first search and single-source shortest path, the vertices in the new active list are a subset of the vertices in $newV_i$. For such algorithms, one can determine whether a vertex is active or not by comparing against its value from the previous superstep V using the function *is_active*. The values of the newly computed vertex list A_i are applied

to the current vertex value array V. The process is repeated for certain number of supersteps or until the active list becomes empty.

Computing $newV_i$ faces the same fine-grained random update problem we discussed in Section 9.3, because the distribution of k_{dst} is irregular.

9.3 Sort-Reduce Algorithm for Sequentializing Updates

The key component of GraFBoost is a sort-reduce accelerator that sequentializes a stream of random updates to an array stored in secondary storage.

Many classes of random update operations can be represented as applying a series of updates to an array, by applying an update function and an argument to an existing value in the array. The series of updates can be represented with a list of key-value pairs, indicating the array location to update and the argument. If we wanted to update an array x and we are given a list xs of key-value pairs $\langle k, v \rangle$ and an update function f, the overall process of applying the updates can be presented as the following:

for each $\langle k, v \rangle$ **in** xs **do**
  $\mathbf{x}[k] = f(\mathbf{x}[k], v)$
end for

For example, to compute the histogram of list of key-value pairs xs, f would be the addition operator, and data in x would be initialized to 0. List xs will contain many key-value pairs $\langle k, v \rangle$, where k is the index of the histogram to add to, and v is the value to add.

Consider performing this computation when both array x and list xs are large, i.e., contain hundreds of millions to billions of elements, and the distribution of keys in xs is extremely irregular with many duplicate keys. This computation would result in many fine-grained random updates to the elements of x, rendering caching ineffective. If x was stored in a secondary storage device, then these fine-grained random access will result in extremely poor use of the disk bandwidth due to granularity mismatch.

One way to make more efficient use of storage bandwidth could be to sort the list xs by key before applying the reduction function f. Once the list is sorted, updates to x, instead of being random, will become completely sequential. If we use an efficient sorting scheme, for example with a high fan-in merge-sorter that reduces the number of sorting passes over xs, the benefits of sequential access usually outweigh the overhead of sorting.

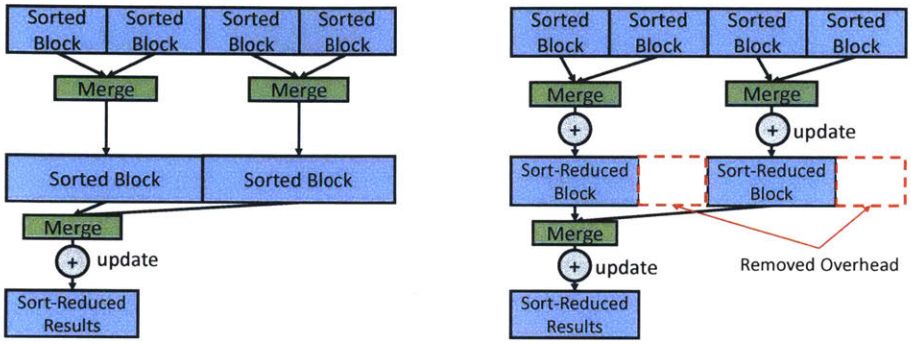
The comparison between sequential access benefits and overhead of sorting can be emphasized with the following proof. Updating 4 byte entries in a 4 KB page results in 1000× I/O amplification. Binary merge-sorting a dataset incurs $log_2 N$ times I/O amplification, where N is the number of elements. In order for the I/O amplification of binary merge-sorting to reach 1000×, the update log xs needs to have more than 2^{1000} elements in it, which is vastly larger number than even the number of atoms in the universe [6].

The relative benefits of sorting become even better using high fan-in merge-sorters. However, while sorting may improve the performance of storage-based systems compared to naïvely using storage as a memory extension, experiments have shown that sorting itself is not enough to make a storage-based system competitive against memory-based systems. The second part of sort-reduce remedies this by further reducing the sorting overhead.

The overhead of the newly introduced sorting phase can be further reduced if the update function f is binary-associative, that is, $f(f(v_1, v_2), v_3) = f(v_1, f(v_2, v_3))$. Given a binary-associative update function, we can use it to merge entries in xs with matching keys, reducing the size of xs. For example, if the entries $\langle k, v_1 \rangle$ and $\langle k, v_2 \rangle$ are discovered in xs, they can be merged into a single entry $\langle k, f(v_1, v_2) \rangle$.

These two methods, which we respectively call *sort* and *reduce*, are leveraged together in the *sort-reduce* method. In this method, merge steps in the sorting algorithm are interleaved with reduction operations to merge key-value pairs for matching keys to reduce the size of xs at each sorting step, as can be seen in Figure 9-2. After each merge step, if two consecutive values have matching keys, they are merged into a single key-value pair before performing the next merge step. We experienced drastic

performance improvements, in terms of reduction of data copied, by interleaving update operations with sort, as we will demonstrate with the graph analytics example in this Chapter.



(a) Completely sorting before applying update

(b) Interleaving sorting and update

Figure 9-2: Interleaving sorting and updates dramatically reduces the amount of partially sorted data to be sorted

Algorithm 2 shows the algorithmic representation of sort-reduce. The algorithm has two parameters k and f, where k is the fan-in of the merge-sorter, and f is the associative update function. Given the input data, which is a list of key-value pairs, the algorithm iteratively performs k-to-1 merge-sort on the input data. After each iteration, the reduce operation is applied to newly merged blocks, in which the update function f is applied to pairs of elements with matching keys. The output is a list of key-value pairs where all keys are unique, and values with the same key have been reduced using the update function.

9.4 Using Sort-Reduce for External Graph Analytics

We can reformulate Algorithm 1 using the sort-reduce method to get Algorithm 3. Algorithm 3 first logs all the updates in list U and feeds it to the SortReduce accelerator to produce $newV_i$. We later show that applying sort-reduce for graph analytics has dramatic performance benefits, and allows GraFBoost to compete with other systems with much larger DRAM capacity.

Algorithm 2 The sort-reduce algorithm

```
▷ Input: input_list of ⟨key, val⟩ elements
list = []
for each e in input_list do                    ▷ Create list with single-element blocks
    list.append([e])
end for

while list.count > 1 do
    new_list = []
    for each k sorted blocks [B_1, ..., B_k] in list do
        sorted_block = merge([B_1, ..., B_k])

        ▷ apply f to pairs with matching keys
        reduced_sorted_block = reduce_f(sorted_block)
        new_list.append(reduced_sorted_block)
    end for
    list = new_list
end while
▷ Output: the only block left in list
```

The associativity restriction for *vertex_update* required by sort-reduce is not very limiting, and it can represent a large amount of important algorithms. It is also a restriction shared with many existing models and platforms, including the Gather-Apply-Scatter model used by PowerGraph [59] and Mosaic [114], as well as the linear algebraic model.

The vertex value V is stored as an array of vertex values in storage, and is accessed using the key as index. Because $newV_i$ and A_i are all sorted by key, access to V is also always in order. As a result, the performance impact of accessing and updating V is relatively low.

In some algorithms where the active list is not a subset of $newV$, the programmer can create a custom function that generates the active vertex list. For example, the active list for the PageRank formulation in Algorithm 4 is the set of vertices which are sources of edges that point to a vertex in $newV$. In this implementation, the next iteration will perform PageRank on the active vertices to update the PageRank of the values in $newV$.

In this implementation, a bloom filter is used to keep track of all vertices which

Algorithm 3 Using sort-reduce for performing a vertex program superstep

▷ A_0 is algorithm init condition
U = [] ▷ Update log is initialize to empty
for each $\langle k_{src}, v_{src} \rangle$ **in** A_{i-1} **do** ▷ Iterate through active vertices
  for each $e(k_{src}, k_{dst})$ **in** G **do** ▷ Iterate through outbound edges of k_{src}
    ev = **edge_program**$(v_{src}, e.prop)$
    U.append($\langle k_{dst}, ev \rangle$) ▷ Edge program results are appended to log
  end for
end for
$newV_i = SortReduce_{vertex\_update}(U)$

triangleright The rest of the algorithm is identical
for each $\langle k, v \rangle$ **in** $newV_i$ **do**
  x = **finalize**$(v, V[k])$ ▷ Apply remaining element-wise calculation
  if is_active$(x, V[k])$ **then** ▷ Whether k should be active at next iteration
    A_i.append($\langle k, x \rangle$)
    $V[k] = v$ ▷ Update value of k
  end if
end for

have edges that point to a vertex in $newV$. $V[k]$ also stores the superstep index i when it was updated, so that sort-reduced values for vertices that were not included in the previous superstep's $newV$ can be ignored. In the interest of performance, bloom filter management and looping through the keyspace can also be implemented inside the accelerator.

9.5 GraFBoost Architecture

Figure 9-3 shows the overall architecture of a hardware-accelerated GraFBoost system. A GraFBoost storage device, which consists of NAND flash storage and a FPGA-based in-storage hardware accelerator, is plugged into a host machine over PCIe, which can be a server, PC or even a laptop-class computer. The GraFBoost storage device uses a custom-designed flash storage hardware [81, 108] to expose raw read, write and erase interfaces of its flash chips to the accelerator and host, instead of through an FTL. All the data structures such as the index and edge lists to describe the graph structure, vertex data V, $newV$, and temporary data structures created and deleted during sort-reduce operation are maintained as files and managed by the

Algorithm 4 Sort-reduce with custom active list generation for PageRank

```
▷ newV_0 is algorithm init condition
bloom = [], U = []
for each ⟨k, v⟩ in newV_{i-1} do
    x = finalize(v, V[k])
    if is_active(x, V[k]) then                                  ▷ If k is active
        for each edge e(k_src, k) in graph G do
            bloom[k_src] = true                  ▷ Set filter for vertices that can affect k
        end for
        V[k_src] = ⟨x, i⟩                                       ▷ Update value of k_src
    end if
end for

for all ⟨k_src, _⟩ in graph G do                                ▷ Loop through keyspace
    if bloom[k_src] = true then          ▷ If k_src may affect previous active vertices
        for each edge e(k_src, k_dst) in graph G do
            ev = edge_program(V[k_src], e.prop)
            U.append(⟨k_dst, ev⟩)
        end for
    end if
end for
newV_i = SortReduce_{vertex_update}(U)
```

host system using a lightweight flash file management system called the Append-Only Flash File System (AOFFS).

The host system executes supersteps one at a time. It initiates each superstep by reading $newV$, parts of V and the graph files, and feeding the information to the edge program. The stream produced by the edge program is fed into the sort-reduce accelerator, which sorts and applies vertex updates to create a $newV$ for the next superstep. We first describe the storage data structures, and then the accelerators involved in graph analytics.

9.5.1 Data Representation

GraFBoost stores graphs in a compressed sparse column format, or outbound edge-list format in graph terminology, as shown in Figure 9-4. For each graph, we keep an index and an edge file in flash storage. For the applications discussed in this chapter, these files do not change during execution. In order to access the outbound edges and their destinations from a source vertex, the system must first consult the

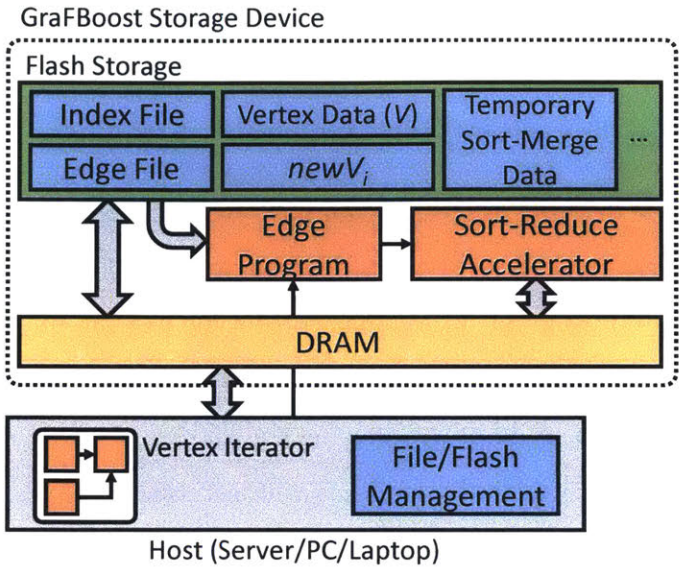


Figure 9-3: The internal components of a GraFBoost system

index file to determine the location of the outbound edge information in the edge file. For some algorithms like PageRank, an inbound edge information is also required, which is stored in the same format. The vertex data file V is stored as a dense array of vertex data. However, the result of the sort-reduce algorithm $newV$, as well as the temporary files created by the sort-reduce accelerators are stored in a sparse representation, which is a list of key-value pairs.

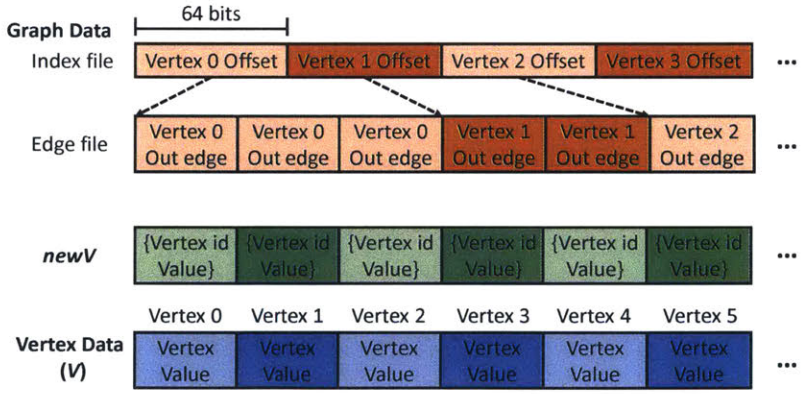


Figure 9-4: Compressed column representation of graph structure

GraFBoost uses the file system AOFFS described in Section 3.2.1. GraFBoost and AOFFS are a good fit because GraFBoost does not require random updates thanks

to sort-reduce.

9.5.2 Effect of Fixed-Width Access to DRAM and Flash

In order to make the best use of DRAM and flash bandwidth at a certain clock speed, the interface to these devices must be wide enough to sustain their bandwidth. For example, the accelerators in our GraFBoost prototype use 256-bit words for communication. We pack as many elements as possible in these 256-bit words, ignoring byte and word alignment, but without overflowing the 256-bit word.

For example, if the key size is 34 bits, it will use exactly 34 bits instead of being individually padded and aligned to 64 bits (see Figure 9-5). Such packing and unpacking entails no overhead in specialized hardware and at the same time it saves a significant amount of storage access bandwidth.

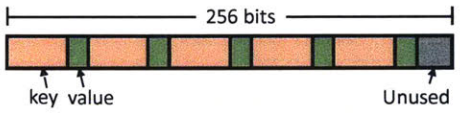


Figure 9-5: Data packing in a 256-bit word

9.5.3 Edge Program Execution

An edge program function is applied to the value of each active vertex and the edge property of each of its outbound edges. GraFBoost is parameterized by edge program function, which itself is parameterized by vertex value type and edge property type. For single-source shortest-path, the edge program adds the vertex and edge values and produces the result as a vertex value. These value types can vary in size from 4 bits to 128 bits. The dataflow involved in edge program execution can be seen in Figure 9-6.

In order to apply the edge program, GraFBoost must first determine the active vertex list for the current iteration, by streaming in $newV$ and corresponding V elements from storage, and using the programmer-provided is_active function. For algorithms that do not fit this model such as PageRank, where the active vertex list

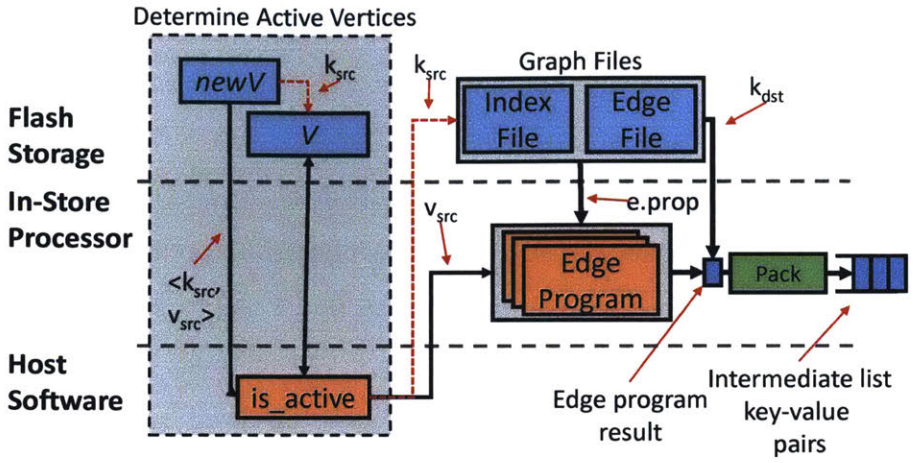


Figure 9-6: Intermediate list generation

is not a subset of $newV$, the programmer can choose to implement custom programs in software that use files in storage and generate a stream of active vertex lists. GraFBoost also provides a *vertex list generator* module in hardware for when all nodes are active with default values. It emits a stream of active vertex key-value pairs with uniform values, and the user can provide the range of keys it generates, as well as the value.

After outbound edges of each active vertex are read from the graph files, they can be fed to the edge program. The GraFBoost system operates an array of parallel edge program instances because the wide interface from flash storage may have many values encoded in it. The resulting values from the edge programs are coupled with the destination vertex index of each edge to create a stream of 256-bit-wide intermediate key-value pairs. Before this stream is emitted, it is packed so that each emitted 256-bit word has as many values in it as possible.

9.5.4 Hardware Implementation of Sort-Reduce

The sort-reduce accelerator operates on the unsorted stream of $\langle k, v \rangle$ pairs coming out of the edge program to create the sorted and reduced $\langle k, v \rangle$ list of newly updated vertex values. The design of the sort-reduce accelerator involves two components: (1) a high-performance hardware-accelerate external sorter, described in detail in

Section 7, and (2) a wire-speed aggregator, which finds consecutive elements with matching keys and applies the update function between them at wire-speed, described in detail in Section 8.

There are three different types of memories of various capacities and speed available for sorting and we exploit the characteristics of each to the fullest extent. These three types of memories are the external flash storage, 1 GB to 2 GB DRAM on the FPGA board, and 4 MB of BRAM inside the FPGA where all the accelerators are implemented. We describe the whole sorting process top down.

We first read an unsorted block into the in-memory buffer with the size of, say 512 MB, from the input stream and sort it in-memory using the DRAM on the FPGA board. We can instantiate as many in-memory sorters as needed to match the memory bandwidth, provided they fit in the FPGA. The size of the sorting block depends on the amount of available DRAM and the number of sorters. The in-memory sorted blocks are streamed through the aggregator accelerator described in Chapter 8, and written to flash. The resulting dataflow for in-memory sort-reduce can be seen in Figure 9-7. This process results in multiple sort-reduced files on the flash storage, where each file is likely to be significantly smaller than the in-memory buffer due to the reduction operation. The actual ratio of reduction for various graph analytics benchmarks is provided in Section 9.7.

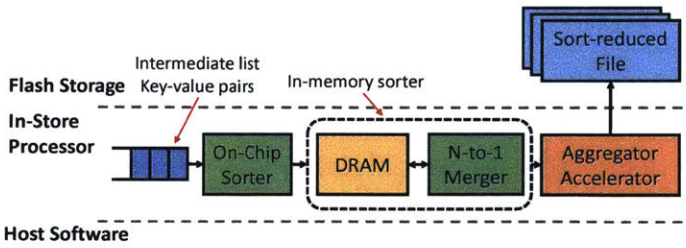


Figure 9-7: Intermediate list is sorted first in on-chip memory granularities and then in DRAM

We then merge k sort-reduced blocks by simultaneously reading each from the flash storage to produce a sorted block of size 16×512 MB, reduce it, and write it to the flash storage. This merger is done in a streaming manner using a k-to-1 merger and DRAM to buffer input and output. In our hardware implementation, the default

fan-in of the merger, k, is 16. It is beneficial to have a large k, because it reduces the number of sort phases that is required to fully sort the input data. As a result, the value of k is chosen to be a large enough value that results in a circuit that fits in the FPGA. The process of reading, merge-reducing and writing blocks is repeated until the full list is sorted. The dataflow for external merge and reduce can be seen in Figure 9-8.

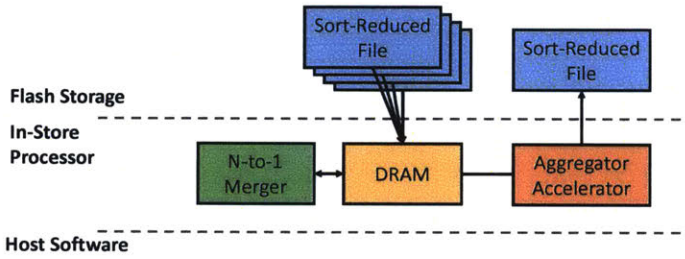


Figure 9-8: External sorting merges sorted chunks in storage into a new sorted chunk in storage

Each level of sort and reduce is performed using the same strategy: data is streamed in either from on-FPGA memory, DRAM, or flash storage, and then it is sorted using a merge-sorter network, streamed through an aggregator accelerator, and streamed back out.

9.6 GraFSoft Implementation

GraFSoft implements all system components in software, and can be run on any off-the-shelf system. GraFSoft shares many design features with the hardware-accelerated GraFBoost system, including the graph data representation. However, it does not implement the bandwidth-efficient packing method introduced in Section 9.5.2, because the benefits of space and bandwidth saved by packing were offset by the software overhead of packing and unpacking data. A fully software implementation allows us to evaluate the benefit of the sort-reduce method by making a fair comparison with other software systems using the same hardware environment, as well as validate our hardware implementation.

The key component in the GraFSoft implementation is the software implementation of sort-reduce. The software sort-reduce implementation is composed of two parts: in-memory sort-reduce and external sort-reduce. The in-memory sort-reduce component maintains a pool of sorter and reducer threads to use available high-performance sorting libraries like `std::sort` to perform sort-reduce on in-memory chunks, and writes the sort-reduced chunks as individual files. The size of the in-memory buffer is parameterized, and is set to 512 MB by default per sort-reducer instance.

The external sort-reduce component maintains a pool of k-to-1 merge-sorters, and repeatedly merges and reduces k files into one using a software merge-sorter until all files have been merged. As in the case of the hardware accelerator case, the default fan-in of the merge-sorter is 16-to-1.

The performance of a single 16-to-1 is important because while many sort-reducer instances can be spawned, eventually the last 16 or less chunks need to be merged into one by a single merger, at the end of the merge tree. Due to the performance limitation of a single thread, a multithreaded 16-to-1 merger is implemented as a tree of 2-to-1 mergers, each running on a separate thread. Figure 9-9 shows the structure of a software merge-reducer. Sorted streams are transferred in large 4 MB chunks in order to minimize inter-process communication overhead.

The first-level nodes (leaves) of the merger tree are actually implemented as a 4-to-1 merge sorter instead of a 2-to-1 merge sorter, since the performance pressure is usually lower at the lower levels. This helps us reduce the thread count of the software sort-reduce implementation, requiring only 8 threads instead of 16.

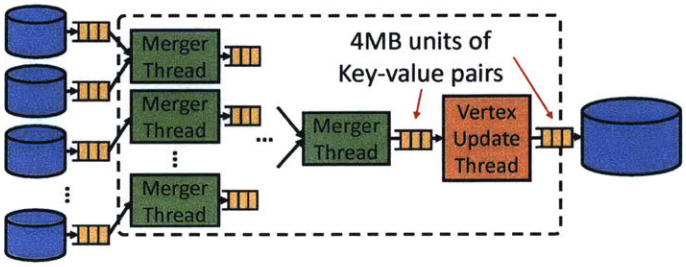


Figure 9-9: A multithreaded software implementation of sort-reduce

We plan to research applying more sophisticated sorting methods to achieve higher GraFSoft performance and lower resource requirements.

9.7 Evaluation

We show the GraFBoost system is desirable for the following two reasons:

1. Using the sort-reduce method in software implementations of large graph analytics (GraFSoft) provides better performance than other in-memory, semi-external, and external systems. Large graphs are defined as that the memory requirements for in-memory and semi-external systems exceeding the available memory of a single node. In this section, we will also show that given a fixed amount of memory, GraFSoft can efficiently process much larger graphs than other systems.
2. The sort-reduce method benefits from hardware acceleration (GraFBoost) both by increased performance and decreased energy consumption. As a result, GraFBoost can not only handle graphs much larger than any other system that we tested, it provides comparable performance even for smaller graphs which other systems are optimized for.

We support these results through evaluations of multiple graph algorithms and graph sizes using various graph analytics frameworks and our own software and hardware implementations of GraFBoost.

9.7.1 Graph Algorithms

The following three algorithms were chosen for evaluation because their implementations were provided by all platforms evaluated. By using the implementations optimized by the platform developers, we are able to do fair performance comparisons.

Breadth-First-Search (BFS): BFS is a good example of an algorithm with sparse active vertices. BFS is a very important algorithm because it forms the basis

and shares the characteristics of many other algorithms such as Single-Source Shortest Path (SSSP) and Label Propagation.

It maintains a parent node for each visited vertex, so that each vertex can be traced back to the root vertex. This algorithm can be expressed using the following vertex program, where $vertexID$ is provided by the system:

function EDGEPROG($vertexValue$,$edgeValue$,$vertexID$)
  return $vertexID$
end function
function VERTEXUPD($vertexValue1$,$vertexValue2$)
  return $vertexValue1$
end function

PageRank (PR): PageRank is a good example of an algorithm with very dense active vertex sets. It also requires a finalize function for dampening, which is often defined as $x = 0.15/NumVertices + 0.85 * v$. In order to remove the performance effects of various algorithmic modifications to PageRank and do a fair comparison between systems, we only measured the performance of the very first iteration of PageRank, when all vertices are active.

PageRank can be expressed using the following program, where $numNeighbors$ is provided by the system:

function EDGEPROG($vertexValue$,$edgeValue$,$numNeighbor$)
  return $vertexValue/numNeighbor$
end function
function VERTEXUPD($vertexValue1$,$vertexValue2$)
  return $vertexValue1$+$vertexValue2$
end function

Betweenness-Centrality (BC): BC is a good example of an algorithm that requires backtracing, which is an important tool for many applications including machine learning and bioinformatics. Each backtracking step incurs random updates to parent's vertex data, which is also handled with another execution of sort-reduce.

The edge and vertex programs for BC is identical to BFS. After traversal is fin-

ished, each generated vertex list's vertex values are each vertex's parent vertex ID. Each list can be made ready for backtracing by taking the vertex values as keys and initializing vertex values to 1, and sort-reducing them. The backtrace sort-reduce algorithm can be expressed with the following algorithm:

function EDGEPROG(*vertexValue*,*edgeValue*,*null*)
  return *vertexValue*
end function
function VERTEXPROG(*vertexValue*1,*vertexValue*2)
  return *vertexValue*1+*vertexValue*2
end function

Once all lists have been reversed and reduced, the final result can be calculated by applying set union to all reversed vertex lists using a cascade of set union operations, with a custom function to add multiply values whenever there is a key match.

9.7.2 Graph Datasets

For each application, we used multiple different graph datasets with different sizes. One important graph dataset we analyzed is the Web Data Commons (WDC) web crawl graph [7] with over 3.5 billion vertices, adding up to over 2 TBs in text format. The WDC graph is one of the largest real-world graphs that are available. Others include synthetic graphs generated at various sizes according to Graph 500 configurations, and the popular Twitter graph. Graph 500 requires graph to be generated using kronecker multiplication [102], which results in power-law graphs. Table 9.1 describes some of the graphs of interest. The size of the dataset is measured after column compressed binary encoding in GraFBoost's format.

name	Twitter	kron28	kron30	kron32	WDC
nodes	41 M	268 M	1 B	4 B	3 B
edges	1.47 B	4 B	17 B	32 B	128 B
edgefactor	36	16	16	8	43
size	6 GB	18 GB	72 GB	128 GB	502 GB
txtsize	25 GB	88 GB	351 GB	295 GB	2648 GB

Table 9.1: Graph datasets that were examined

9.7.3 Compared Graph Analytics Systems

We evaluate the performance of GraFSoft and GraFBoost compared to various prominent graph analytics systems in the in-memory, semi-external and external categories. In the in-memory category, we compared against distributed GraphLab [109], which is one of the most prominent distributed in-memory systems. In the semi-external category, we compared against X-Stream [151] and FlashGraph [190], which are the fastest semi-external software systems we could find. In the fully external category, we compared against GraphChi [95], which is the most prominent of the very few fully external graph analytics systems.

9.7.4 Evaluation with Graph with Billions of Vertices

To show the effectiveness of the sort-reduce method in software (GraFSoft) and the hardware acceleration of that method (GraFBoost), we compared their performances against semi-external and external graph analytics frameworks for large graphs. To perform this evaluation, we use a single server with 32 cores of Intel Xeon E5-2690 (2.90 GHz) and 128 GB of DRAM. For graph storage, the server is equipped with five 512 GB PCIe SSDs with a total of 6 GB/s of sequential read bandwidth. All software systems including GraFSoft were executed on this system.

GraFBoost was implemented on the BlueDBM platform [81], where a hardware-accelerated storage system was plugged into the PCIe slot of a server with 24-core Xeon X5670 and 48 GBs of memory. The storage device consists of a Xilinx VC707 FPGA development board equipped with 1 GB DDR3 SODIMM DRAM card, and augmented with two 512 GB NAND-flash expansion cards, adding up to 1 TB of capacity. Each flash expansion card is capable of delivering 1.2 GB/s read performance and 0.5 GB/s write performance, while the DRAM card can deliver 10 GB/s. We observed that the biggest performance bottleneck of GraFBoost is the single slow DIMM card, and also compare against the projected performance of GraFBoost2, equipped with DRAM capable of 20 GB/s.

To start the evaluation, we explore the performance of our three algorithms on a

synthesized scale 32 kronecker graph, and the largest real-world graph we could find, the WDC web crawl graph.

Evaluation on these graphs shows that both GraFSoft and GraFBoost are the only systems among those tested that can achieve high performance in all algorithms on both graphs. All other systems either ran out of memory and failed, or took too long to finish in some or all benchmarks.

9.7.4.1 Performance on Kron32

Figure 9-10 shows the performance of various systems running graph algorithms on the Kron32 graph. All performances have been measured by taking the inverse of the elapsed time to complete each algorithm execution, and normalized to the performance of GraFSoft (red horizontal line).

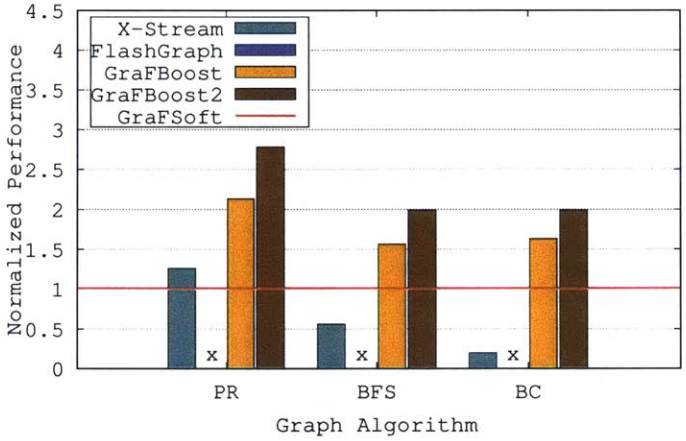


Figure 9-10: Performance of the algorithms on the Kronecker 32 graph, on a machine with 128 GB of memory, normalized to the performance of software GraFBoost **(Higher is faster)**

The hardware-accelerated GraFBoost system achieves the highest performance across all algorithms, and even GraFSoft (horizontal red line) demonstrated faster performance than all compared systems in two of three algorithms. Of the various systems compared against, the semi-external X-Stream was the only system other than GraFBoost that was able to execute any of the three algorithms on Kron32. The in-memory GraphLab was unable to accommodate such a large graph in the

128 GB of available memory, and was unable to execute any algorithm. The external system GraphChi did not finish any algorithm in a reasonable amount of time. The semi-external FlashGraph was also unable to accommodate the vertex data in the available memory, and was unable to execute any algorithm. The hardware-accelerated GraFBoost was able to deliver 57 MTEPS (Million Traversed Edges Per Second) on BFS.

Table 9.2 shows the elapsed time in seconds for each graph algorithm on each system tested.

Algorithm	X-Stream	GraFSoft	GraFBoost	GraFBoost2
PR	914	1157	542	415
BFS	1627	919	587	462
BC	5712	1152	705	577

Table 9.2: Elapsed time in seconds to execute PageRank, Breadth-First-Search and Betweenness-Centrality on Kron32

9.7.4.2 Performance on the WDC Graph

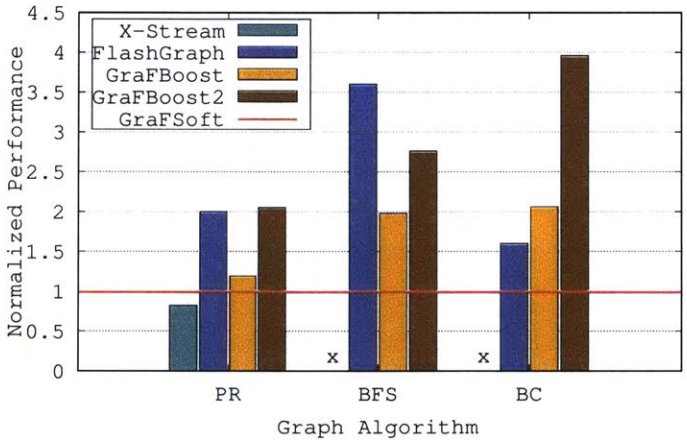


Figure 9-11: Performance of the algorithms on the WDC graph, on a machine with 128 GB of memory, normalized to the performance of software GraFBoost **(Higher is faster)**

Figure 9-11 shows the performance of various systems running graph algorithms on the WDC graph, normalized against GraFSoft performance. As before, all performances have been measured by taking the inverse of the elapsed time to complete each

algorithm execution, and normalized to the performance of GraFSoft (red horizontal line).

For the WDC graph, FlashGraph was able to demonstrate high performance for PageRank and BFS, but X-Stream was unable to finish within the time limit for BFS and BC. Because the WDC graph has much fewer vertices compared to Kron32 (3 billion vs. 4 billion), FlashGraph could accommodate all vertex data in the 128 GB of available memory. As a result, FlashGraph was often able to deliver faster performance compared to GraFBoost. On the other hand, the diameter of the WDC graph is large compared to Kron32 (over 5,000 vs. 8), resulting in low performance of X-Stream in BFS and BC due to the limitations of the system in handling sparse active vertices. X-Stream processes all vertices at every superstep, resulting in about 500 seconds of processing time per superstep. As a result, the total processing time of X-Stream for BFS and BC was projected to exceed two million seconds, or 23 days. In contrast, BFS on Kron32 required only 8 supersteps, resulting in much better performance on X-Stream (Figure 9-10). GraFBoost delivered 75 MTEPS on BFS.

Table 9.3 shows the elapsed time in seconds for each system to finish execution the algorithms tested on the WDC graph. As in the case with Kron32, we were not able to measure the performance of GraphChi or GraphLab, due to low performance and insufficient memory, respectively.

Algorithm	X-Stream	FlashGraph	GraFSoft	GraFBoost	GraFBoost2
PR	1640	674	1348	1131	655
BFS	X	699	2495	1258	903
BC	X	1980	3118	1509	786

Table 9.3: Elapsed time in seconds to execute PageRank, Breadth-First-Search and Betweenness-Centrality on WDC

It can be seen from the two large graphs that the GraFBoost systems are the only ones capable of delivering consistently high performance with terabyte-scale graphs on the given machine. This strength becomes even more pronounced when graphs are even larger relative to available memory size, which we will analyze in detail in the following section.

9.7.4.3 Performance with Larger Graphs Relative to Memory Capacity

We also evaluated the performance impact of graph sizes relative to available memory, by processing large graphs on systems with variable memory capacities. As graph sizes relative to memory capacity became larger, the GraFBoost systems quickly became the only systems capable of maintaining high performance.

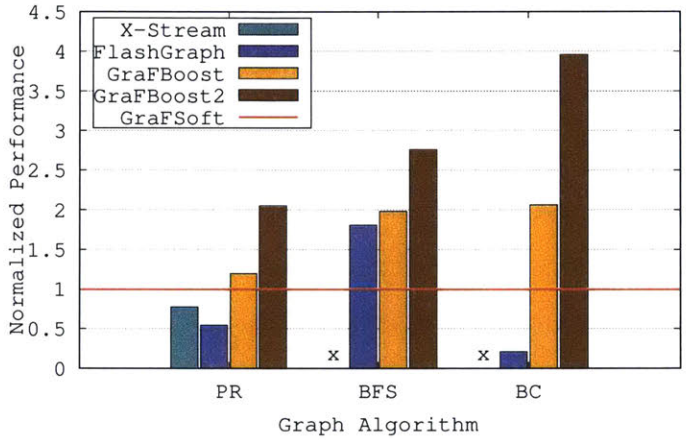


Figure 9-12: Performance of the algorithms on a machine with 64 GB of memory, normalized to the performance of software GraFBoost **(Higher is faster)**

Figure 9-12 shows the performance of systems working on the WDC graph on a reasonably affordable machine with 32 Xeon cores and 64 GBs of memory (instead of previous 128 GB), as normalized to the performance of GraFSoft. We can see that the hardware-accelerated GraFBoost implementations perform better than all systems compared. Even GraFSoft is faster than both X-Stream and FlashGraph, except for FlashGraph on BFS. FlashGraph performance on breadth-first-search is still relatively high, because BFS has a relatively sparse active vertex set during execution, and FlashGraph could still accommodate most vertex data in memory.

The same as the previous two experiments with large graphs, the in-memory system GraphLab was unable to run any algorithm due to insufficient memory, and GraphChi was unable to finish execution in a reasonable amount of time. This section presents experimental results from five systems that manage to finish more than one experiment: FlashGraph, X-Stream, GraFSoft, GraFBoost, and GraFBoost2.

Below, we analyze the performance impact of memory capacity for different graph

algorithms. Memory capacity is denoted in percentage of vertex data size, where 100% is the space required to store 2^{32} 8-byte values, or 32 GB.

PageRank Figure 9-13 shows the elapsed time of each system running PageRank on the WDC graph on a system with various memory capacities. GraFBoost takes consistently small amount of time regardless of available memory capacity because its memory requirement is lower (16 GB or less for GraFSoft, megabytes for GraFBoost).

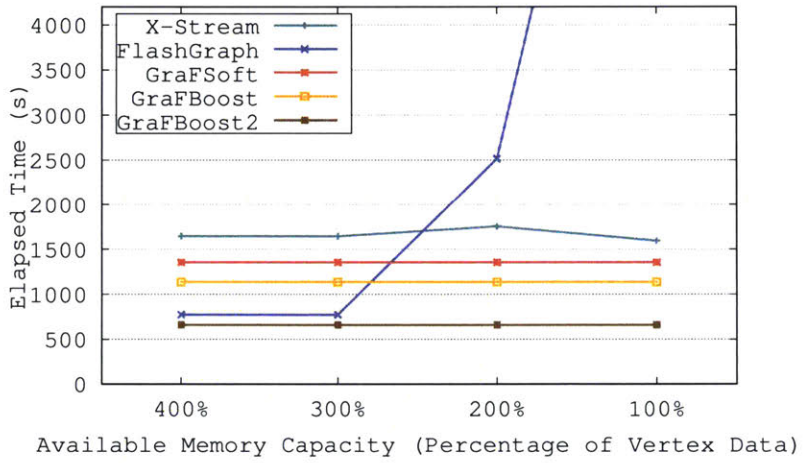


Figure 9-13: PageRank iteration execution time on machines with various amounts of memory **(Lower is faster)**

When memory capacity of 300% or more is available, FlashGraph showed the fastest performance with the exception of the projected GraFBoost2 system. But its performance degrades sharply and quickly becomes the slowest at memory capacities of 200% or less, as the problem size becomes larger compared to memory capacity, causing swap thrashing.

X-Stream showed performance slower than FlashGraph when enough memory was available, but it was able to maintain performance with smaller memory by dividing the graph into partitions on systems with smaller memory in order to fit each partition in available memory. One caveat was that the vertex update logs between partitions became too large to fit in our flash array, and required us to install more storage.

Breadth-First Search Figure 9-14 shows the elapsed time of each system running breadth-first-search on the WDC graph on a system with various memory capacities.

Because the active vertex list during BFS execution is usually small relative to the number of vertices in the graph, the memory requirements of a semi-external system like FlashGraph are also relatively low. As a result, FlashGraph performance degrades slower compared to the PageRank experiment.

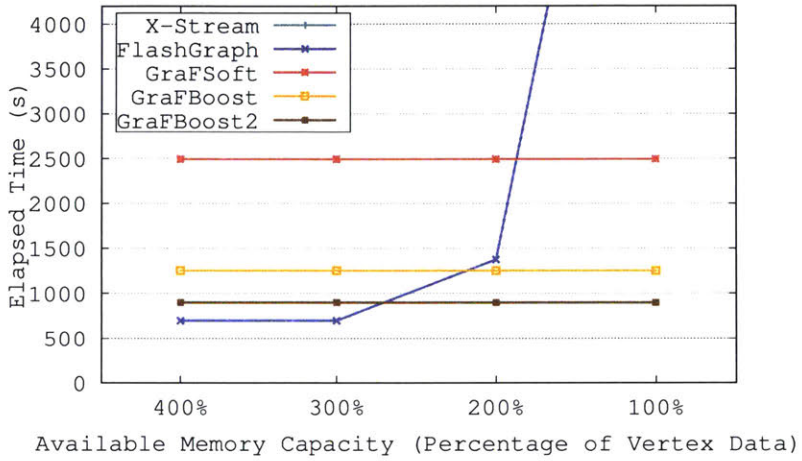


Figure 9-14: Breadth-First-Search execution time on machines with various amounts of memory **(Lower is faster)**

On systems with memory capacity of 300% or more, FlashGraph shows the fastest performance of the systems tested. It starts thrashing swap space when available memory drops below 200% of vertex data size, and becomes even slower than GraFSoft. Data for X-Stream is not available on this figure, because we were unable to measure the performance of X-Stream on the WDC graph in a reasonable amount of time for any configuration, due to the same reason described in Section 9.7.4.2.

Betweenness-Centrality Figure 9-15 shows the elapsed time for various systems running betweenness-centrality on the WDC graph. The memory requirements of BC is higher than PageRank and BFS, resulting in faster performance degradation for FlashGraph. We were unable to measure the performance of X-Stream for BC on the WDC graph for the same reason as with BFS.

These experimental results show that performance degradation of some semi-external systems can drop sharply as graph sizes become larger relative to available memory capacity. In situations when it is costly to provision safely large capacity of

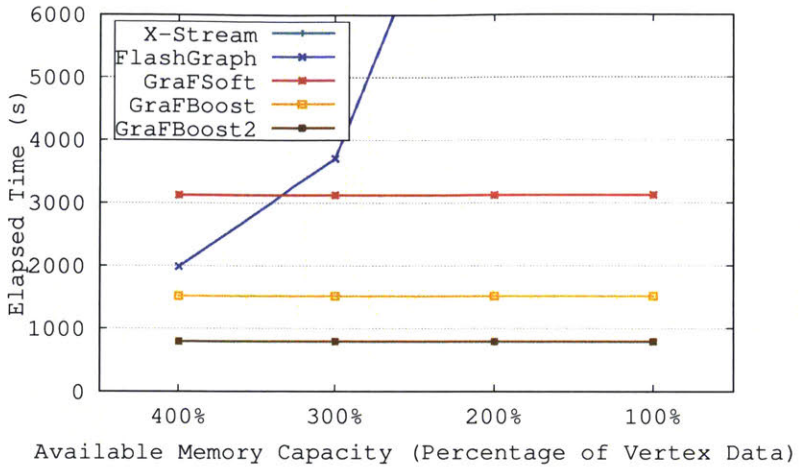


Figure 9-15: Betweenness-Centrality execution time on machines with various amounts of memory **(Lower is faster)**

memory, GraFBoost or GraFSoft becomes a very attractive alternative.

9.7.5 Small Graph Evaluations (Less Than 1 Billion Vertices)

The target application of GraFBoost is graphs that are too large to comfortably fit in the main memory of reasonable systems. Smaller graphs can be efficiently managed by conventional semi-external or in-memory systems, and are not the interest of this work. Ideally, a graph analytics system should dynamically choose between conventional or sort-reduce-based methods based on the size of the graph or active list. We include experimental results with smaller graphs here, in order to demonstrate that the performance of sort-reduce-based systems does not suffer greatly compared to other systems even when the graph is small enough for other systems to handle comfortably. This means the choice for GraFBoost against in-memory or semi-external systems can be made aggressively, because the performance penalty for running in-memory or semi-external systems with slightly less memory than required is much harsher than the performance penalty for running GraFBoost when sufficient memory is available.

Software evaluations were done using the same server with 32 cores and 128 GB of memory, but were only provided with one SSD instead of five. GraFBoost also used only one flash card instead of two, matching 512 GB capacity and 1.2GB/s

bandwidth. At the risk of doing an unfair comparison, we also evaluated a 5-node cluster of distributed GraphLab (GraphLab5) using systems with 24-core Xeon X5670 and 48 GB of memory networked over 1G Ethernet.

Across all experiments, GraphLab could not handle graphs larger than the smallest Twitter graph, and even the distributed GraphLab with five nodes could not handle graphs larger than Kron28, which is the second smallest. Larger graphs could only be handled by semi-external or external systems.

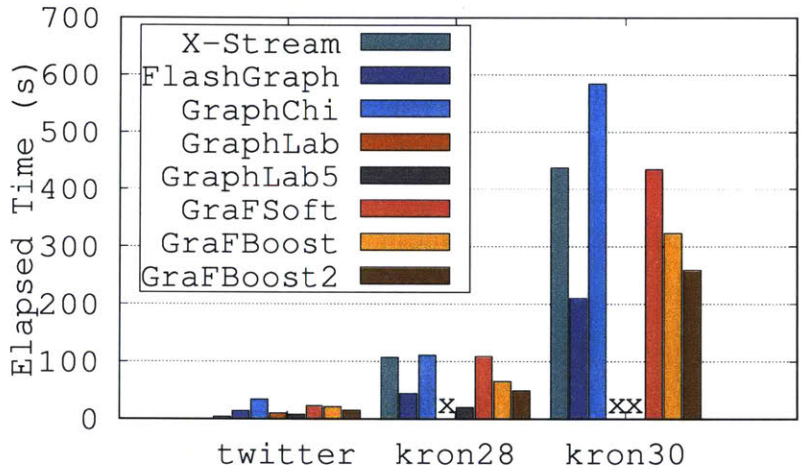


Figure 9-16: Execution time of PageRank on small graphs **(Lower is faster)**

	X-Stream	FlashGraph	GraphChi	GraphLab	GraphLab5	GraFSoft	GraFBoost	GraFBoost2
Twitter	3.6	14	33.7	10	8	23	21	15
kron28	107	44	111	X	20	109	65	49
kron30	438	210	584	X	X	435	323	259

Table 9.4: Execution time of PageRank on small graphs in seconds

Figure 9-16 shows the processing time for PageRank on smaller graphs with 1 billion or less vertices. The same data in exact numbers is provided in Table 9.4. For PageRank on the smallest Twitter data, X-Stream demonstrated the fastest performance, even faster than the in-memory systems GraphLab and GraphLab5. X-Stream performs cache optimizations for small data, and effectively becomes a single-node in-memory system. As graph sizes become larger, performance of X-Stream degrades quickly, compared to in-memory and other semi-external systems. Besides X-Stream on the Twitter data, the distributed GraphLab system achieved the highest performance, followed by the single-node GraphLab, for graphs it could handle. The

semi-external FlashGraph and hardware-accelerated GrapFBoost systems were next in terms of performance, followed by GraFSoft and X-Stream. The external system GraphChi was the slowest in performance.

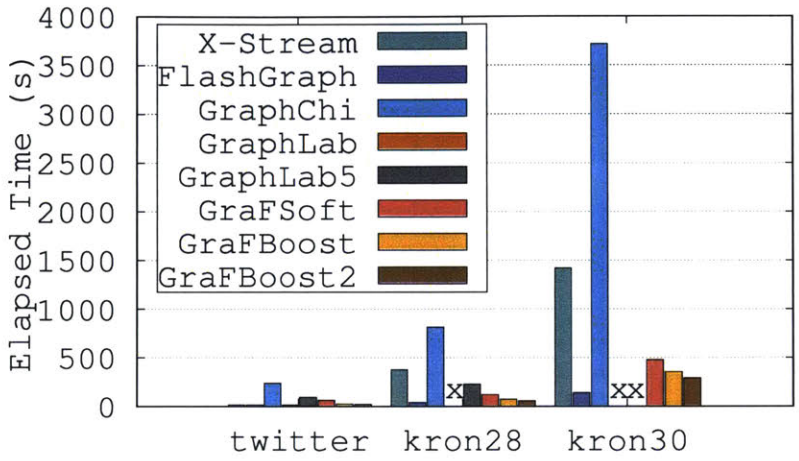


Figure 9-17: Execution time of BFS on small graphs **(Lower is faster)**

	X-Stream	FlashGraph	GraphChi	GraphLab	GraphLab5	GraFSoft	GraFBoost	GraFBoost2
Twitter	16	16	239	15	95	62	26	20
kron28	376	42	815	X	229	122	74	58
kron30	1422	140	3717	X	X	477	354	291

Table 9.5: Execution time of BFS on small graphs in seconds

Figure 9-17 shows the processing time for BFS on smaller graphs with 1 billion or less vertices. The same data is presented in exact numbers in Table 9.5. While GraphLab5 demonstrates the best performance for PageRank on Kron28, it is relatively slow for BFS, even against single-node GraphLab for the Twitter data, as well as GraphLab, GraFSoft, and GraFBoost. This is most likely due to the network becoming the bottleneck with irregular data transfer patterns. FlashGraph, GraFSoft and GraFBoost systems followed closely in terms of performance, while X-Stream's performance was relatively lower, as the active list became sparse. As with PageRank, GraphChi achieved the slowest performance among systems tested.

For small graphs, the relative performance of GraphBoost systems is not as good as with bigger graphs, but is comparable to the fastest systems. When datasets are small, the effectiveness of cacheing graph edge data in semi-external systems increases, and sort-reduce becomes an unnecessary overhead.

9.7.5.1 Further Discussion of PageRank

Figure 9-18 shows comparisons of full executions of PageRank, because some systems like Galois do not operate in terms of supersteps, and instead generate work asynchronously. Systems were either run until convergence, or failing that, executed for 20 iterations. The results that are not our own measurements are taken from McSherry, 2015 [122], and are marked with an asterisk (*). The performances for Spark, Giraph, GraphX and GraphLab are from a 8-node cluster with a total of 128 cores, and the *SingleThread* system shows the performance of a single-threaded in-memory implementation running on a high-end laptop. In-memory system Galois was also run on the 8-node cluster with 128 cores.

Some distributed systems performed poorly due to their distribution overhead, while X-Stream performed very fast because with such a small dataset, it can optimize its cache usage and effectively become a single-node in-memory system.

We must be aware that both due to different machine configurations and organization of computation, the numbers shown in Figure 9-18 cannot be simply taken at face value. Instead, it should only be taken as an estimate of how a simple deployment of each system performs.

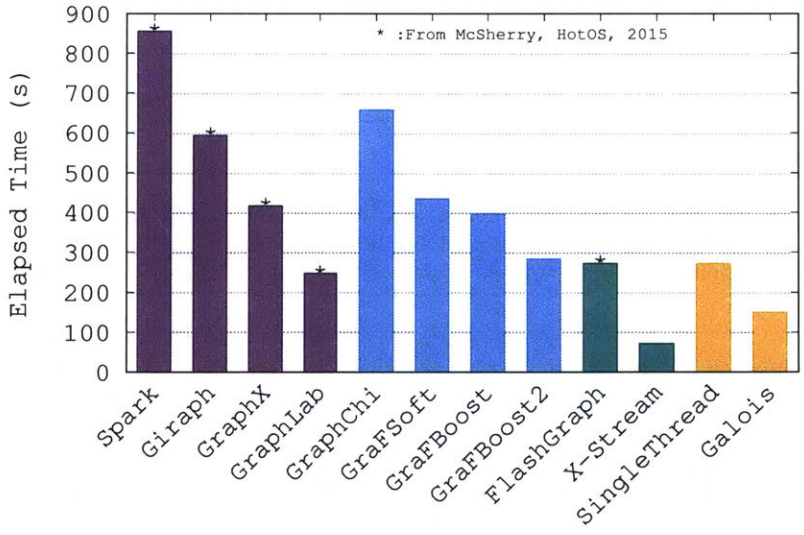


Figure 9-18: Execution time of PageRank on the Twitter graph until convergence

9.7.6 Hardware and Software Performance Comparison

In our evaluations, the hardware-accelerated GraFBoost invariably demonstrated higher performance compared to the GraFSoft software implementation. There are two reasons for this performance difference: the performance benefits of the sort-reduce hardware accelerator, and the benefits of the BlueDBM flash interface.

9.7.6.1 Performance Benefits of the Sort-Reduce Accelerator

We compared the performance of the software sort-reduce implementation against the hardware accelerator implementation. The performance was measured by executing a single merge-reduce phase on multiple gigabytes of data. Table 9.6 describes the configurations we used to measure performance. Because the performance of the software sort-reducer varied according to the distribution of the update log, we tested three different distributions. The hardware accelerator performance was constant regardless of distribution.

Name	Description
SWSame	Software, all keys are the same
SWRand	Software, keys are random
SWUniq	Software, all keys are unique
HW	Hardware

Table 9.6: Systems configurations to evaluate sort-reduce performance

Figure 9-19 shows that a single instance of the sort-reduce accelerator generally achieves more than 4 times the performance compared to the software implementation. While more software sort-reducers can be instantiated to take advantage of modern multi-processors, their effects are limited due to the number of cores available. As each software sort-reducer in our implementation instantiates eight threads, a 32-core machine can run up to four instances of the software sort-reducer in parallel, which is still likely to be slower than a single hardware instance. Furthermore, instantiating four sort-reducers to compete with a single hardware accelerator would involve all 32 threads solely working on sort-reduce, not leaving much system resources for other useful work. Of course, more hardware sort-reduce accelerators can

be instantiated to achieve better performance as well.

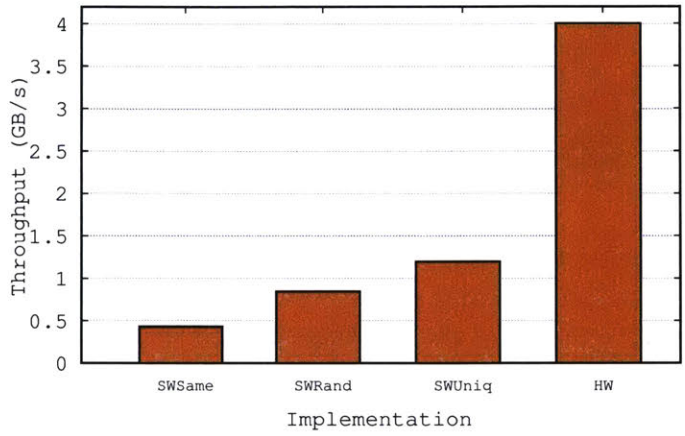


Figure 9-19: Sort-reduce hardware accelerator performance comparison against a software implementation

This performance difference can be seen in the PageRank example in Figure 9-11, where differences in flash access performance are suppressed because the graph adjacency list access is very regular, and performance is determined by the performance of sort-reduce.

9.7.6.2 Performance Benefits of the BlueDBM flash interface

GraFBoost is able to make consistently high performance random accesses in 8 KB page granularity, by using the BlueDBM high-performance flash interface. In contrast, GraFSoft uses off-the-shelf flash storage devices, which are optimized for coarser granularity accesses. This access granularity issues manifest in performance while reading the edge file to run the edge program. When the active vertex list is sparse, reading the edge file in coarse granularities results in wasted bandwidth, but trying to read finer granularities from an off-the-shelf storage results in slower I/O. This performance difference is emphasized in the BFS and BC examples in Figure 9-11, where the access pattern is relatively sparse.

9.7.7 Benefits of Interleaving Reduction with Sorting

One of the biggest factors of GraFBoost's high performance is the effectiveness of interleaving reduction operations with sorting. Figure 9-20 shows the ratio of data that was merge-reduced at every phase compared to if reduction was not applied, starting from the intermediate list generated when all nodes are active. The number of merge-reduce phases required until the intermediate data is completely merged depends on the size of the graph.

The reduction of data movement is significant, especially in the case of the two real world graphs, the Twitter graph and WDC graph. In these graphs, the size of the intermediate list has already been reduced by over 80% and 90% even before the first write to flash. This reduces the amount of total writes to flash by over 90%, minimizing the impact of sorting and improving flash lifetime.

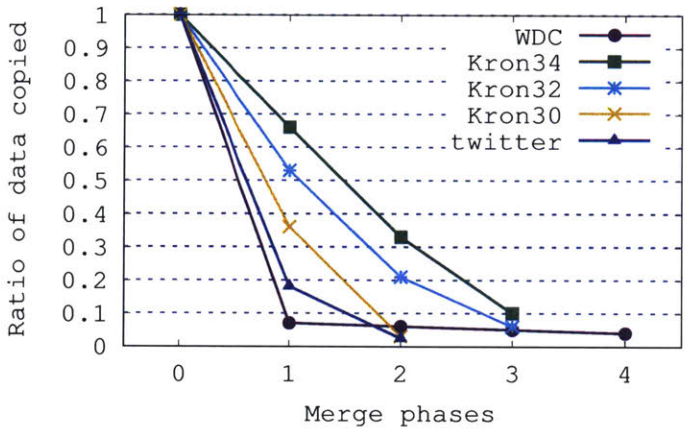


Figure 9-20: The fraction of data that is written to storage after each merge-reduce phase

9.7.8 System Resource Utilization

Table 9.7 shows the typical system resource utilizations of the systems compared, while running at full capacity PageRank on the WDC graph. Both FlashGraph and X-Stream attempted to use all of the available 32 cores' CPU resources. It is most likely because these systems spawn a lot more threads than GraFBoost. As a result,

both systems record 3200% CPU usage while running. The software GraFBoost tries to use all of the available processing resources, but does not exceed 1800% because performance is usually bound by storage performance. The hardware-accelerated GraFBoost does not require much processor resources because the resource-intensive sort-reduce has been offloaded to the hardware accelerator.

Name	Memory	Flash Bandwidth	CPU
GraFBoost	2 GB	1.2 GB/s	200%
GraFSoft	8 GB	500 MB/s*, 4 GB/s**	1800%
FlashGraph	60 GB	1.5 GB/s	3200%
X-Stream	80 GB	2 GB/s	3200%

*During intermediate list generation
**During external merge-reduce

Table 9.7: Typical system resource utilization during PageRank on WDC

9.7.8.1 Power Consumption

There are two sources of power efficiency in GraFBoost: flash-based analytics and hardware acceleration. By using flash storage instead of DRAM, we are able to use a single node, or a much smaller cluster of machines to handle very large graphs, reducing the system power budget. And by using hardware acceleration to offload most of graph analytics computation, we can use a machine with much less processing resources, further reducing power consumption. Figure 9-21 shows the measured power consumption for the three system configurations we used to evaluate systems: The five-node cluster for in-memory GraphLab, the server with 128 GB of DRAM, and GraFBoost. These three configurations correspond to requiring a large cluster to accommodate the whole graph in-memory, using software sort-reduce to process graphs in flash, and using hardware accelerators with sort-reduce, respectively.

Our prototype implementation of GraFBoost with hardware acceleration consumes about 160 W of power, including the host server as well as the accelerated flash storage. This is much less than our server setup used to run software systems including FlashGraph, GraFSoft and others, which consumed over 410W during operation. The distributed in-memory system with five nodes consumed even more

power.

The power consumption of GraFBoost can be further reduced by replacing the mostly idle host server, which is consuming 110 W of power, with a low-power embedded server with less processing resources. Using an embedded server with 40 W power budget will bring down the whole system power consumption to 90 W without sacrificing performance. Software systems like GraFSoft cannot reduce power consumption by reducing computation resources, because high computation resources are critical for performance.

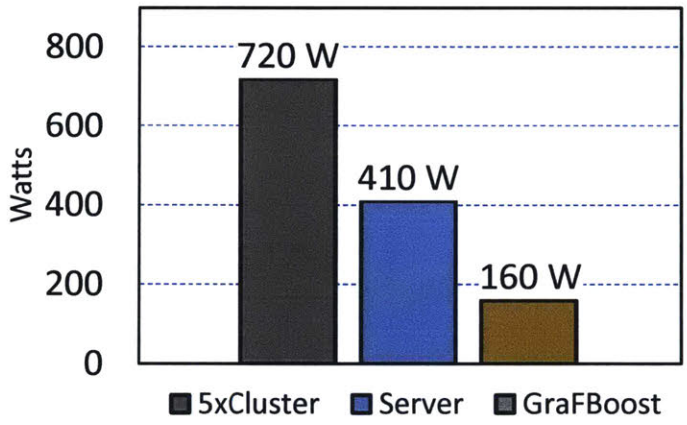


Figure 9-21: Power consumptions of the three machine configurations are measured

9.8 Chapter Summary

In this chapter, we have presented GraFBoost, an external graph analytics system that uses the novel sort-reduce algorithm to perform high-speed analytics on graphs with billions of vertices, on an affordable machine with very small memory and high performance flash. We presented two flavors of GraFBoost: GraFSoft, which is implemented entirely in software and can run on any commodity machine, and GraFBoost, which implements the sort-reduce algorithm as a hardware accelerator in the in-storage reconfigurable accelerator on the flash storage device. Using large graphs with billions of vertices, we demonstrated that wimpy PC-class machine coupled with a GraFBoost accelerated storage can achieve server-class performance while main-

taining a much lower power footprint. Compared to state-of-the-art graph analytics software, GraFBoost with both hardware and software implementation of sort-reduce were the only systems that could continue to achieve high performance with all terabyte-scale benchmarks on the system provided. Even on smaller graphs the other systems were optimized for, GraFBoost achieved comparable performance at a fraction of power and resource budget.

Graph analytics using sort-reduce requires the vertex update function to be an associative binary function, which is a restriction shared with many other models and systems including the linear algebraic systems and Gather-Apply-Scatter model. Our GraFBoost implementations assume the edge file to be static, but it is not a fundamental limitation of the system design. One of our future research directions is managing algorithms that change the edge file, which will still use sort-reduce to update vertex data.

It is possible that by using more sophisticated methods for external sorting, the performance of GraFSoft can be improved further, and one of our immediate research goals is software methods for faster performance. With faster software performance including not only faster sort-reduce but faster fine-grained storage access, GraFSoft may also be able to saturate flash bandwidth and achieve performance comparable to GraFBoost. However, GraFBoost will always have the benefit of lower power consumption using hardware acceleration.

Although our novel sort-reduce algorithm was initially designed for external graph analytics, it will be useful for any Big Data application that requires fine-grained random updates. Near-term future plans include applying sort-reduce for bioinformatics algorithms to lower the cost of analytics.

Chapter 10

Conclusion

It is simply accepted in the modern world that collecting massive amounts of data and applying data scientific and machine learning methods on them will yield deep insight that were previously unavailable. Analyzing so-called "Big Data" has become so prevalent that the term now is becoming superfluous. However, the cost of acquiring and operating a high-performance system to perform complex analytics on such large amounts of data is often a limiting factor. One of the fundamental causes of high cost is the requirement of large amounts of memory, caused by the fine-grained random access requirement of many analytics algorithms. In this thesis, we explored a novel system architecture using cheaper flash storage coupled with in-storage reconfigurable hardware accelerators to lower the cost of Big Data analytics without sacrificing performance. By organizing the system so that flash bandwidth is not wasted, and by removing computational bottlenecks using hardware accelerators, we were able to demonstrate that such a system can perform competitively against much costlier in-memory systems.

In this chapter, we introduce the summary of the contributions in this thesis, and outline some future research directions.

10.1 Thesis Summary

We first presented a new system architecture for Big Data analytics using flash storage and in-storage hardware accelerators called BlueDBM. BlueDBM is a rack-scale cluster system where each node is installed with BlueDBM storage devices, which are equipped with in-storage hardware accelerators, as well as a separate storage area network directly between storage devices. The in-storage accelerator has low latency access to storage, and has enough computation capacity to saturate the performance of storage, network, and the DRAM buffer on the storage device. The separate storage area network provides low enough latency access to distributed storage that the whole storage cluster can be treated as a uniform address space. Bypassing the host and networking the storage devices directly enables BlueDBM to easily focus the bandwidth of multiple storage devices onto a single in-storage accelerator. The BlueDBM storage device also allows the in-storage accelerator or the host server high-performance low-level access into flash storage by having an option of removing or bypassing the flash translation layer.

We implemented a 20-node BlueDBM prototype cluster to provide empirical proof of the benefits of this architecture. Because no commercial off-the-shelf storage device was equipped with the storage features BlueDBM required, we constructed a custom flash storage device, which we call minFlash. A minFlash storage device included a Xilinx VC707 FPGA accelerator, 1 GB of on-board DRAM buffer, four 20 Gbps serial communication links pinned out as SATA ports, as well as 1 TB of NAND flash storage, capable of 2.4 GB/s reads and 2 GB/s writes. We first demonstrated using microbenchmarks, that our prototype implementation was making efficient use of all of its system components, including the storage device and network.

Our prototype BlueDBM cluster has enabled research into a large array of applications, and showed that for all applications surveyed, there were important scenarios in which a flash-based system with in-storage hardware accelerators can compete with, or outperform much costlier in-memory cluster systems.

The first application was approximate high-dimensional nearest-neighbor search,

which involved coarse-grained random access into a large dataset, and computing the distance against a query data. Our experience with this application taught us two lessons: (1) when computation overhead is moderate to large, as seen with document similarity and image similarity metrics, the benefits of hardware-accelerated distance metrics over multithreaded software implementations were high enough for flash-based systems to become competitive against in-memory software systems. (2) Even when computation overhead was minimal, as seen with the hamming distance metric, the performance of the in-memory system drops sharply when the memory capacity is not enough, and even a small fraction of memory accesses spill over into swap space. Additionally, the low power consumption of flash and FPGA accelerators resulted in a much better power-performance compared to a DRAM-based system.

The second application was terabyte-scale sort, which involved streaming I/O from and to flash storage, as well as computation involved in fast external merge-sort. One of the novel designs for this application was the use of a library of four types of merge-sorters for the four types of memory fabric available on the system: on-chip registers, on-chip block RAM, on-board DRAM, and flash storage. By making the maximum use of flash storage bandwidth as well as the small (1 GB) amount of on-board DRAM, using a sorting-network-based parallel merge-sorter, our single-node implementation was able to demonstrate high performance comparable to a 21-node Hadoop cluster. The most impressive feature of this implementation, however, is its power efficiency. Despite the inefficient design of our storage device prototype, and the high idle power of our underused host server, a single node prototype was able to achieve better power-performance compared to the current JouleSort champion. The projected power-performance of a single-node system with a power-efficient embedded processor is over double that of the JouleSort champion.

The third application was hardware-acceleration of wire-speed database aggregation operations, which are used for queries like GROUPBY. The design of the aggregator accelerator solved two issues that were previously unsolved in the area: aggregation functions with multicycle latency, and multiple elements per cycle ingestion rate. As a result, a single instance of our accelerator implementation, using a

floating point multiplier with 7 cycles of latency as the aggregator function, was able to saturate flash, PCIe and network available on our BlueDBM prototype system by ingesting four or more elements every cycle. This application did not involve flash storage, but is designed as an independent accelerator that can be plugged into other systems. In fact, it was an integral component of the graph analytics platform, which was the next application we explored.

The fourth application introduced in this thesis was terabyte-scale graph analytics, which involves fine-grained random updates, or read-modify-writes. We were able to efficiently remove the random access from graph analytics using a novel algorithm called sort-reduce. Sort-reduce takes a stream of update requests and sorts it by the update index, while performing reduction operations in-place whenever a pair of requests with a matching index is discovered. Both the merge-sorter accelerator and the aggregator accelerator implemented for the previous applications played an important part in the high-performance implementation of the sort-reduce accelerator. The sort-reduce hardware accelerator removed the computation bottleneck from the sort-reduce portion of the graph analytics implementation, and allowed it to make the best use of storage and memory performance. We call our graph analytics platform GraFBoost, and we compared its performance against various state-of-the-art in-memory, semi-external, and external graph analytics software. While all other systems either ran out of memory and failed after a certain graph size, or were generally too slow, GraFBoost was able to perform competitively against the fastest systems, while handling graphs larger than what any other system could handle.

There were more applications and research enabled by the BlueDBM platform led by our colleagues and are not described in detail in this thesis. Prominent examples include a file system with cross-layer optimization for performance and flash lifetime, and a flash-based memcached-like key-values cache.

Our new flash-optimized file system called Application-managed Log-structured File System (ALFS) moves most flash management functionality from the FTL into the application, to make intelligent decisions about erasure, hot pages and more. As a result, ALFS was able to improve performance by 80%, and flash lifetime by 38%.

Another benefit of moving flash management out of the storage device was reducing the size of DRAM on the FTL, as memory requirements for mapping could be reduced by a factor of 128.

The key-value cache, called BlueCache, optimizes the network and database management using the BlueDBM storage area network and hardware acceleration in order to reclaim the performance difference between flash and DRAM. As a result, BlueCache was able to outperform a prominent software flash-based KVS by over four times, and even outperformed in-memory systems when the in-memory system has more than 7.4% misses.

10.2 Future Work

Our future research direction also mainly focuses on making complex analytics of large amounts of data more affordable, using hardware-accelerated flash storage. First of all, we plan to continue researching on applications that may benefit from accelerated flash storage, as well as a general platform for complex analytics using accelerated flash storage devices. Secondly, we plan to explore alternative system architectures for effectively using flash storage and hardware accelerators.

10.2.1 More Applications

We plan to continue exploring more applications that may benefit from flash storage, hardware acceleration and fast networks. One of the immediate target applications is the Somatic MUtation FINder (SMUFIN) [124], an algorithm and implementation for discovering somatic mutations from patients that may have been responsible for the illness. This application involves a large amount of sequenced genome data, as well as complex computation on them. We are collaborating with the Barcelona Supercomputing Center, the developers of this algorithm, to explore how it can benefit from accelerated flash storage. One of the low-hanging fruits is the first stage of the algorithm, K-mer counting, which is a good fit for the sort-reduce algorithm we developed for the graph analytics application.

We plan to expand into more general areas in bioinformatics, and one of the goals in that direction is accelerating, or lowering the cost of running the Genome Analysis ToolKit (GATK) [121].

10.2.2 General Platform for Accelerated Flash Storage

One of the important goals in researching on many applications that benefit from accelerated flash storage, is to identify a suite of tools or algorithms that is required for a more general platform for performing complex analytics. Sort-reduce will be one of such a suite of tools. Once enough tools have been identified to collectively cover a wide range of important applications, we will be able to determine an effective programming model or software system to best use them. Much like how MapReduce or Spark was able to make distributed processing available for a wide population of programmers, we hope to provide a way for more people to enjoy the benefits of flash-based external analytics and hardware acceleration.

10.2.3 Alternative Architecture for Accelerated Flash Storage

While the BlueDBM has enabled us to conduct many important research projects, we believe we need to continue researching on alternative architectures that may better suit important applications and the modern datacenter environment. One such architecture may be disaggregated accelerated storage, where a large storage appliance coupled with multiple hardware accelerators is installed as a single entity, and accessed by all machines in a datacenter over a fast network. Disaggregated storage paradigms have been gaining traction due to their ease of management and the availability of fast networks. Coupling in-storage hardware accelerators with disaggregated storage may make the technology more available for wide-scale datacenter use. Other architectures we are exploring include cache-coherent hardware accelerators, accelerators in the network datapath, as well as simply as a PCIe-attached accelerator peripheral.

Bibliography

[1] Amazon ec2 f1 instances. https://aws.amazon.com/ec2/instance-types/f1/. Accessed: 2017-12-31.

[2] Graph CREST. http://opt.imi.kyushu-u.ac.jp/graphcrest/eng/. Accessed: 2017-11-21.

[3] Intel fpga acceleration hub - platforms. https://www.altera.com/solutions/acceleration-hub/platforms.html. Accessed: 2017-12-31.

[4] Intel fpgas power acceleration-as-a-service for alibaba cloud. https://newsroom.intel.com/news/intel-fpgas-power-acceleration-as-a-service-alibaba-cloud/. 2017-10-12 (Accessed: 2018-04-12).

[5] Neo4j: The world's leading graph databse. https://neo4j.com/. Accessed: 2017-11-17.

[6] Number of atoms in the universe. https://educationblog.oup.com/secondary/maths/numbers-of-atoms-in-the-universe. 2015-11-24 (Accessed: 2018-4-17).

[7] Web data commons - hyperlink graphs. http://webdatacommons.org/hyperlinkgraph/. Accessed: 2017-10-01.

[8] *TeraSort Benchmark Comparison for YARN*, 2014 (Accessed January 18, 2017).

[9] Mahdi Abdelguerfi and Arun K Sood. A fine-grain architecture for relational database aggregation operations. *IEEE Micro*, 11(6):35–43, 1991.

[10] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[11] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 105–117. IEEE, 2015.

[12] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[13] Bruno Angelucci, R Fantechi, G Lamanna, E Pedreschi, R Piandani, J Pinzino, M Sozzi, F Spinella, and S Venditti. The fpga based trigger and data acquisition system for the cern na62 experiment. *Journal of Instrumentation*, 9(01):C01055, 2014.

[14] Infiniband Trade Association. *Infiniband*, 2018 (Accessed April 12, 2018).

[15] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.

[16] 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*, pages 44–54. ACM, 2006.

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

[18] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[19] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest. Maxwell - a 64 fpga supercomputer. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 287–294, Aug 2007.

[20] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *Computer vision–ECCV 2006*, pages 404–417. Springer, 2006.

[21] A. Becher, F. Bauer, D. Ziener, and J. Teich. Energy-aware sql query acceleration through fpga-based dynamic partial reconfiguration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014.

[22] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[23] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10gbps line-rate key-value stores with fpgas. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, 2013. USENIX.

[24] A Boccardi, M Gasior, OR Jones, KK Kasinski, and RJ Steinhagen. The fpga-based continuous fft tune measurement system for the lhc and its test at the cern sps. In *Particle Accelerator Conference, 2007. PAC. IEEE*, pages 4204–4206. IEEE, 2007.

[25] Aydın Buluç and John R Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, page 1094342011403516, 2011.

[26] T. Bunker and S. Swanson. Latency-optimized networks for clustering fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 129–136, April 2013.

[27] 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.

[28] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[29] Adrian M. Caulfield and Steven Swanson. Quicksan: A storage area network for fast, distributed, solid state disks. *SIGARCH Comput. Archit. News*, 41(3):464–474, June 2013.

[30] 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, pages 181–192, New York, NY, USA, 2009. ACM.

[31] Hsinchun Chen, Roger HL Chiang, and Veda C Storey. Business intelligence and analytics: from big data to big impact. *MIS quarterly*, pages 1165–1188, 2012.

[32] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. When apache spark meets fpgas: a case study for next-generation dna sequencing acceleration. In *The 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[33] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. When apache spark meets fpgas: a case study for next-generation dna sequencing acceleration. In *The 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[34] Derek Chiou. The microsoft catapult project. In *Workload Characterization (IISWC), 2017 IEEE International Symposium on*, pages 124–124. IEEE, 2017.

[35] Benjamin Y Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. Xsd: Accelerating mapreduce by harnessing the gpu inside an ssd. In *Proceedings of the 1st Workshop on Near-Data Processing*, 2013.

[36] 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*, pages 91–102. ACM, 2013.

[37] Yuk-Ming Choi and Hayden Kwok-Hay So. Map-reduce processing of k-means algorithm with fpga-accelerated computer cluster. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 9–16. IEEE, 2014.

[38] Eric S Chung, John D Davis, and Jaewon Lee. Linqits: Big data on little clients. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 261–272. ACM, 2013.

[39] Eric S. Chung, James C. Hoe, and Ken Mai. Coram: An in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 97–106, New York, NY, USA, 2011. ACM.

[40] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5):332–343, 2009.

[41] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 217–226, New York, NY, USA, 2017. ACM.

[42] Jason Dai. Toward efficient provisioning and performance tuning for hadoop. *Proceedings of the Apache Asia Roadshow*, 2010:14–15, 2010.

[43] Yoginder S Dandass, Shane C Burgess, Mark Lawrence, and Susan M Bridges. Accelerating string set matching in fpga hardware for bioinformatics research. *BMC bioinformatics*, 9(1):197, 2008.

[44] Christian De Schryver. *FPGA Based Accelerators for Financial Applications.* Springer, 2015.

[45] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern*

Recognition, 2009. CVPR 2009. IEEE Conference on, pages 248–255. IEEE, 2009.

[46] C. Dennl, D. Ziener, and J. Teich. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 45–52, April 2012.

[47] C. Dennl, D. Ziener, and J. Teich. Acceleration of sql restrictions and aggregations through fpga-based dynamic partial reconfiguration. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28, April 2013.

[48] 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*, pages 1221–1230. ACM, 2013.

[49] 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.

[50] Andreas Ebert. *NTOSort*, 2013 (Accessed January 11, 2017).

[51] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM, 2011.

[52] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 251–262, New York, NY, USA, 1999. ACM.

[53] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[54] X. Fang, S. Misra, G. Xue, and D. Yang. Smart grid - the new and improved power grid: A survey. *IEEE Communications Surveys Tutorials*, 14(4):944–980, Fourth 2012.

[55] A. Farmahini-Farahani, A. Gregerson, M. Schulte, and K. Compton. Modular high-throughput and low-latency sorting units for FPGAs in the Large Hadron Collider. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 38–45, June 2011.

[56] A. Farmahini-Farahani, H. J. Duwe III, M. J. Schulte, and K. Compton. Modular Design of High-Throughput, Low-Latency Sorting Units. *IEEE Transactions on Computers*, 62(7):1389–1402, July 2013.

[57] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind, and Joel Emer. Leveraging latency-insensitivity to ease multiple fpga design. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 175–184, New York, NY, USA, 2012. ACM.

[58] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[59] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30, 2012.

[60] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 325–336, New York, NY, USA, 2006. ACM.

[61] Thomas Graves. *GraySort and MinuteSort at Yahoo on Hadoop 0.23*, 2013 (Accessed January 11, 2017).

[62] Jim Gray, Chris Nyberg, Mehul Shah, and Naga Govindaraju. *Sort Benchmark Home Page*, 2016 (Accessed January 11, 2017).

[63] Anthony Gregerson, Amin Farmahini-Farahani, Ben Buchli, Steve Naumov, Michail Bachtis, Katherine Compton, Michael Schulte, Wesley H Smith, and Sridhara Dasu. Fpga design analysis of the clustering algorithm for the cern large hadron collider. In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 19–26. IEEE, 2009.

[64] NVMHCI Work Group. *NVM Express*, 2018 (Accessed April 12, 2018).

[65] 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.

[66] PK Gupta. Accelerating datacenter workloads. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[67] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th International Symposium on Microarchitecture*. ACM, 2016.

[68] Sergej Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. Noftl: Database systems on ftl-less flash storage. *Proceedings of the VLDB Endowment*, 6(12):1278–1281, 2013.

[69] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. Rekindling network protocol innovation with user-level stacks. *SIGCOMM Comput. Commun. Rev.*, 44(2):52–58, April 2014.

[70] Hanaa M Hussain, Khaled Benkrid, Huseyin Seker, and Ahmet T Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 248–255. IEEE, 2011.

[71] Dylan Hutchison, Jeremy Kepner, Vijay Gadepally, and Adam Fuchs. Graphulo implementation of server-side sparse matrix multiply in the Accumulo database. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–7. IEEE, 2015.

[72] Gordon Inggs, David Thomas, and Wayne Luk. A heterogeneous computing framework for computational finance. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 688–697. IEEE, 2013.

[73] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[74] Hiroshi Inoue and Kenjiro Taura. SIMD- and Cache-friendly Algorithm for Sorting an Array of Structures. *Proc. VLDB Endow.*, 8(11):1274–1285, July 2015.

[75] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 35:1–35:35, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

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

[77] Michael Johannes Jaspers. *Acceleration of read alignment with coherent attached FPGA coprocessors*. PhD thesis, PhD thesis, TU Delft, Delft University of Technology, 2015.

[78] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. Unlocking ordered parallelism with the swarm architecture. *IEEE Micro*, 36(3):105–117, 2016.

[79] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, Berkeley, CA, USA, 2014. USENIX Association.

[80] Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao, Mark R Nutter, and Jeremy D Schaub. *Tecent Sort*, 2016 (Accessed January 11, 2017).

[81] 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 *ISCA*, pages 1–13. ACM, 2015.

[82] 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):7:1–7:31, June 2016.

[83] Seok-Hoon Kang, Dong-Hyun Koo, Woon-Hak Kang, and Sang-Won Lee. A case for flash memory ssd in hadoop applications. *International Journal of Control and Automation*, 6(1), 2013.

[84] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–12. IEEE, 2013.

[85] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, September 1998.

[86] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 137–146, New York, NY, USA, 2003. ACM.

[87] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. Graphs, matrices, and the graphblas: Seven good reasons. *Procedia Computer Science*, 51:2453–2462, 2015.

[88] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[89] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[90] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage processing of database scans and joins. *Information Sciences*, 327:183–200, 2016.

[91] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[92] Dirk Koch and Jim Torresen. FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on Fpgas for Large Problem Sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM.

[93] 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.

[94] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[95] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[96] 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. USENIX Association.

[97] S. Lee, J. Kim, and Arvind. Refactored design of i/o architecture for flash storage. *Computer Architecture Letters*, PP(99):1–1, 2014.

[98] S. Lee, J. Kim, and A. Mithal. Refactored design of i/o architecture for flash storage. *IEEE Computer Architecture Letters*, 14(1):70–74, Jan 2015.

[99] 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.

[100] Sungjin Lee, Ming Liu, Sang Woo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *FAST*, pages 339–353. USENIX Association, 2016.

[101] Hans-Otto Leilich, Günther Stiege, and Hans Christoph Zeidler. A search processor for data base management systems. In *Fourth International Conference on Very Large Data Bases, September 13-15, 1978, West Berlin, Germany.*, pages 280–287, 1978.

[102] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 133–145. Springer, 2005.

[103] Isaac TS Li, Warren Shum, and Kevin Truong. 160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga). *BMC bioinformatics*, 8(1):185, 2007.

[104] Jeff W Lichtman, Hanspeter Pfister, and Nir Shavit. The big data challenges of connectomics. *Nature neuroscience*, 17(11):1448–1454, 2014.

[105] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 159–164, Oct 2014.

[106] Zhongduo Lin and Paul Chow. Zcluster: A zynq-based hadoop cluster. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 450–453. IEEE, 2013.

[107] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 295–304, New York, NY, USA, 2003. ACM.

[108] Ming Liu, Sang-Woo Jun, Sungjin Lee, Jamey Hicks, et al. minflash: A minimalistic clustered flash array. In *DATE*, pages 1255–1260. IEEE, 2016.

[109] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[110] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, pages 340–349, 2010.

[111] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[112] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[113] Xiaoyu Ma, Dan Zhang, and Derek Chiou. Fpga-accelerated transactional execution of graph workloads. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 227–236, New York, NY, USA, 2017. ACM.

[114] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *EuroSys*. ACM, 2017.

[115] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 363–374. IEEE, 2015.

[116] Grzegorz Malewicz, Matthew H Austern, Aart J.C. Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[117] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces. In *Similarity Search and Applications*, pages 132–147. Springer, 2012.

[118] R. Marcelino, H. C. Neto, and J. M. P. Cardoso. Unbalanced FIFO sorting for FPGA-based systems. In *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*, pages 431–434, Dec 2009.

[119] Vivien Marx. Biology: The big challenges of big data, 2013.

[120] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, October 2015.

[121] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[122] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[123] Inc. Micron Technology. *Micron Advances Persistent Memory with 32GB NVDIMM*, 2017 (Accessed April 11, 2018).

[124] Valentí Moncunill, Santi Gonzalez, Sílvia Beà, Lise O Andrieux, Itziar Salaverria, Cristina Royo, Laura Martinez, Montserrat Puiggròs, Maia Segura-Wang, Adrian M Stütz, et al. Comprehensive characterization of complex structural variations in cancer by directly comparing genome sequence reads. *Nature biotechnology*, 32(11):1106, 2014.

[125] S.W. Moore, P.J. Fox, S.J.T. Marsh, A.T. Markettos, and A. Mujumdar. Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 133–140, April 2012.

[126] Rene Mueller, Jens Teubner, and Gustavo Alonso. Glacier: a query-to-hardware compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1159–1162. ACM, 2010.

[127] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting Networks on FPGAs. *The VLDB Journal*, 21(1):1–23, February 2012.

[128] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(11):2227–2240, 2014.

[129] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, and Houman Homayoun. Accelerating big data analytics using fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 164–164. IEEE, 2015.

[130] Netflix. *Ephemeral Volatile Caching in the cloud*, 2012 (Accessed May 10, 2018).

[131] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[132] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *nsdi*, volume 13, pages 385–398, 2013.

[133] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. GraphGen: An FPGA framework for vertex-centric graph computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 25–28. IEEE, 2014.

[134] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge Path - Parallel Merging Made Simple. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1611–1618, Washington, DC, USA, 2012. IEEE Computer Society.

[135] Yasin Oge, Masato Yoshimi, Takefumi Miyoshi, Hideyuki Kawashima, Hidetsugu Irie, and Tsutomu Yoshinaga. An efficient and scalable implementation of sliding-window aggregate operator on fpga. In *Computing and Networking (CANDAR), 2013 First International Symposium on*, pages 112–121. IEEE, 2013.

[136] Aleph One. Yaffs: Yet another flash file system, 2002.

[137] Mark Oskin, Frederic T Chong, and Timothy Sherwood. *Active pages: A computation model for intelligent memory*, volume 26. IEEE Computer Society, 1998.

[138] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, et al. The case for ramcloud. *Communications of the ACM*, 54(7):121–130, 2011.

[139] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 211–218. IEEE, 2017.

[140] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 166–177. IEEE, 2016.

[141] Esen A. Ozkarahan, Stewart A. Schuster, and Kenneth C. Smith. RAP - an associative processor for database management. In *American Federation of Information Processing Societies: 1975 National Computer Conference, 19-22 May 1975, Anaheim, CA, USA*, pages 379–387, 1975.

[142] Aaron Parsons, Don Backer, Chen Chang, Daniel Chapman, Henry Chen, Pierre Droz, Christina De Jesus, David Macmahon, Andrew Siemion, Dan Werthimer, et al. A new approach to radio astronomy signal processing: packet switched, fpga-based, upgradeable, modular hardware and reusable, platform-independent signal processing libraries. In *In: Proceedings of the XXXth General Assembly of the International Union of Radio Science*. Citeseer, 2006.

[143] Aaron Parsons, Donald Backer, Chen Chang, Daniel Chapman, Henry Chen, Patrick Crescini, Christina De Jesus, Chris Dick, Pierre Droz, David MacMahon, et al. Petaop/second fpga signal processing for seti and radio astronomy.

In *Signals, Systems and Computers, 2006. ACSSC'06. Fortieth Asilomar Conference on*, pages 2031–2035. IEEE, 2006.

[144] Greg Pass and Ramin Zabih. Comparing images using joint histograms. *Multimedia systems*, 7(3):234–240, 1999.

[145] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[146] 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, and Xiao Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *SIGARCH Comput. Archit. News*, 42(3):13–24, June 2014.

[147] M Wasi-ur Rahman, NS Islam, X Lu, J Jose, H Subramoni, H Wang, and DK Panda. High-performance rdma-based design of hadoop mapreduce over infiniband. In *International Workshop on High Performance Data Intensive Computing (HPDIC), in conjunction with IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[148] Md Wasi-ur Rahman, Xiaoyi Lu, Nusrat Sharmin Islam, and Dhabaleswar K. (DK) Panda. Homr: A hybrid approach to exploit maximum overlapping in mapreduce over high performance interconnects. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 33–42, New York, NY, USA, 2014. ACM.

[149] Suzanne Rivoire, Mehul A Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 365–376. ACM, 2007.

[150] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[151] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[152] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference*

on File and Storage Technologies, FAST'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[153] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[154] Ashok Samal and Prasana A Iyengar. Automatic recognition and analysis of human faces and facial expressions: A survey. *Pattern recognition*, 25(1):65–77, 1992.

[155] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, May 2009.

[156] 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.

[157] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[158] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013.

[159] David Sidler, Zsolt Istvan, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. doppiodb: A hardware accelerated database. SIGMOD, New York, NY, USA, 2017. ACM.

[160] 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.

[161] W. Song, D. Koch, M. Lujn, and J. Garside. Parallel Hardware Merge Sorter. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 95–102, May 2016.

[162] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database analytics acceleration using fpgas. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 411–420. ACM, 2012.

[163] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database analytics acceleration using fpgas. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 411–420, New York, NY, USA, 2012. ACM.

[164] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database analytics acceleration using fpgas. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 411–420. ACM, 2012.

[165] A. Theodore Markettos, P.J. Fox, S.W. Moore, and A.W. Moore. Interconnect for commodity fpga clusters: Standardized or customized? In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[166] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[167] Kuen Hung Tsoi and Wayne Luk. Axel: A heterogeneous cluster with fpgas and gpus. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 115–124, New York, NY, USA, 2010. ACM.

[168] N. Tsuda, T. Satoh, and T. Kawada. A pipeline sorting chip. In *1987 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 270–271, Feb 1987.

[169] Twitter. *twitter/fatcache: Memcache on SSD.*

[170] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, 2016. USENIX Association.

[171] Jiamang Wang, Yongjun Wu, Hua Cai, Zhipeng Tang, Zhiqiang Lv, Bin Lu, Yangyu Tao, Chao Li, Jingren Zhou, and Hong Tang. *FuxiSort*, 2015 (Accessed January 11, 2017).

[172] Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. Replication-based fault-tolerance for large-scale graph processing. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 562–573. IEEE, 2014.

[173] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on

the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 11:1–11:12, New York, NY, USA, 2016. ACM.

[174] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[175] Tim S. Woodall, Galen M. Shipman, George Bosilca, Richard L. Graham, and Arthur B. Maccabe. High performance rdma protocols in hpc. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 76–85, Berlin, Heidelberg, 2006. Springer-Verlag.

[176] L. Woods, Z. Istvan, and G. Alonso. Hybrid fpga-accelerated sql query processing. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–1, Sept 2013.

[177] Louis Woods, Zsolt Istvan, and Gustavo Alonso. Hybrid fpga-accelerated sql query processing. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–1. IEEE, 2013.

[178] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: an intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, 2014.

[179] Nathan A Woods and Tom VanCourt. Fpga acceleration of quasi-monte carlo in finance. In *Field programmable logic and applications, 2008. FPL 2008. International Conference on*, pages 335–340. IEEE, 2008.

[180] 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.

[181] Xilinx. *Floating-Point Operator.*

[182] Reynold Xin, Parviz Deyhim, Ali Ghodsi, Xiangrui Meng, and Matei Zaharia. *GraySort on Apache Spark by Databricks*, 2014 (Accessed January 11, 2017).

[183] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[184] Jennifer Xu and Hsinchun Chen. Criminal network analysis and visualization. *Communications of the ACM*, 48(6):100–107, 2005.

[185] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, et al. Bluecache: A scalable distributed flash-based key-value store. *Proceedings of the VLDB Endowment*, 10(4):301–312, 2016.

[186] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[187] Kjell ystein Arisland, Anne Cathrine Aasb, and Are Nundal. VLSI parallel shift sort algorithm and design. *Integration, the VLSI Journal*, 2(4):331 - 347, 1984.

[188] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[189] Yue Zhao, Kenji Yoshigoe, Mengjun Xie, Suijian Zhou, Remzi Seker, and Jiang Bian. Lightgraph: Lighten communication in distributed graph-parallel processing. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 717–724. IEEE, 2014.

[190] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[191] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 487–495. Curran Associates, Inc., 2014.

[192] Kaile Zhou, Chao Fu, and Shanlin Yang. Big data driven smart energy management: From big data to big insights. *Renewable and Sustainable Energy Reviews*, 56:215–225, 2016.

[193] Shuheng Zhou et al. Gemini: Graph estimation with matrix variate normal instances. *The Annals of Statistics*, 42(2):532–562, 2014.

[194] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, 2015. USENIX Association.