Terabyte Sort on FPGA-Accelerated Flash Storage

Sang-Woo Jun, Shuotao Xu, Arvind
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
mail: {wjun, shuotao, arvind}@csail.mit.edu

***Abstract*—Sorting is one of the most fundamental and useful applications in computer science, and continues to be an important tool in analyzing large datasets. An important and challenging subclass of sorting problems involves sorting terabyte scale datasets with hundreds of billions of records. The conventional method of sorting such large amounts of data is to distribute the data and computation over a cluster of machines. Such solutions can be fast but are often expensive and power-hungry. In this paper, we propose a solution based on 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. We show that by appropriate use of accelerators we can remove all the computation bottlenecks so that the end-to-end sorting performance is limited only by the flash storage bandwidth. We demonstrate that our flash-based system matches the performance of existing distributed-cluster solutions of much larger scale. More importantly, our prototype is able to show almost twice the power efficiency compared to the existing Joulesort record holder. An optimized system with less wasteful components is projected to be four times more efficient compared to the current record holder, sorting over 200,000 records per joule of energy.**

I. Introduction

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 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 is ever increasing due to the increasing size of the datasets of interest. As a result, sorting can easily become a performance bottleneck.

If the datasets does not fit in the available memory of a single machine then it must be stored either in secondary storage, or distributed across multiple machines, or both. In such a setting, sorting algorithms need to be modified to compensate for the access-latency of the secondary storage or network. Many sorting systems mitigate these overheads by using fast storage, such as SSDs or RAID, and fast networks, such as 10Gbps Ethernet or Infiniband. The fastest systems for sorting terabyte scale data today are built on a cluster of machines that use a MapReduce processing platform such as Hadoop.

The high computational overhead of sorting has prompted research into sorting accelerators. Such efforts include parallel sorting algorithms for multiprocessors and sorting algorithms that take advantage of SIMD instructions such as Intel's Streaming SIMD Extensions (SSE) [1], [2]. There is also much research into sorting accelerators on larger scale SIMD appliances such as General Purpose Graphic Processing Units (GPGPU) [3], [4]. Application-specific sorting hardware via Field Programmable Gate Arrays (FPGA) [5]–[10] or Application Specific Integrated Circuits (ASIC) [11]–[13] are also under active investigation.

This paper proposes a system architecture for merge-sorting terabyte-size datasets using an FPGA-accelerated flash storage. This work is an important component for a larger in-store accelerator platform for a graph analytics system we are building. The high performance accelerators on the FPGA ensures that the computational limitations of sorting are effectively removed, and is replaced by the limitations of flash and DRAM bandwidth.

Our system includes different types of hardware sorting accelerators which operate at different levels of granularity. The basic unit of sorting is a *tuple*, which consists of N entities that can be packed into the width of the datapath. In our implementation the datapath width is 256 bits. Depending on the workload, N can be 2 (128 bit entities) to 8 (32 bit entities). Since N is a small number, it can be efficiently sorted using a sorting network [14] in a pipelined and parallel manner. Larger granularities are sorted using a collection of *N-tuple mergers* which are organized to efficiently use the memory hierarchy. Each merger emits N sorted values at every cycle using the ideas from a recently proposed FPGA merge-sorter [5]. The following four sorting accelerators are used by our system:

- *Tuple Sorter* : Sorts an N-tuple using a sorting network.
- *Page Sorter* : Sorts an 8KB (a flash page) chunk of sorted N-tuples in on-chip memory.
- *Super-Page Sorter* : Sorts 16 8K-32MB sorted chunks in DRAM.
- *Storage-to-Storage Sorter* : Sorts 16 512MB or larger sorted chunks in flash.

The design of the various sorting networks and merge-sort accelerators described above is not new. Our contribution is how we organized various hardware sorters to make the best use of a memory hierarchy composed of flash, DRAM and on-chip BRAM. The Super-Page size is derived from the available hardware; given larger DRAM, the Super-Page sorter could sort larger chunks and improve the performance of the system by reducing the number of times the data is copied back and forth from flash. We show that our design has substantial power, cost and performance benefits over existing software-centric systems.

We have implemented a prototype with all the sorters and support infrastructure on a Xilinx VC707 FPGA development board coupled with a Xeon server, using a flash expansion card plugged into the FMC port of the VC707 board. We evaluated the performance of our system using randomly generated terabyte-size datasets, composed of ten to hundred billion records.

Measuring the sorting performance of a prototype implementation with a single accelerated storage showed that such a system was able to match the performance of cluster systems of much larger scale, at a fraction of cost and power budget. For example, one node was able to sort a 1TB dataset of over 10 billion key-value pairs from the Terasort benchmark in 700 seconds. This is more than half the performance of a published 21-node MapR cluster [15]. The performance of our system can be doubled simply by adding another accelerated storage device, which will allow our system to exceed the performance of the MapR cluster with a single node.

The most notable characteristic of our system is its power efficiency. For the Terasort/Joulesort [16] benchmark data, a single node instance is capable of sorting 1TB of 16 byte records, or over 68 billion records, in less than 5,000 seconds while consuming less than 140W of power. This translates to over 100,000 records sorted per joule of energy consumed, which is almost double the current record holder in the Joulesort benchmark. 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. The power efficiency is particularly impressive considering almost 100W of power is consumed by the host server, which is not doing much useful work. We also present a system design with an embedded processor and two storage devices, which is expected to sort over 200,000 records per joule, almost four times the current record holder performance.

The rest of the paper is organized as follows: Existing literature related to large scale sorting is explored in Section II. The detailed architecture and implementation of our system is described in Section III. We present the performance evaluation of our implementation in Section IV, and conclude in Section V.

II. Related Work

A. Large Scale Sorting

Sorting has been one of the most widely studied research topics since the dawn of computer science. Appropriate sorting algorithms have to be chosen for different types of data and available hardware resources. With the advances in parallel processors, it is vital to parallelize sorting algorithms for performance but it is not straightforward to do so.

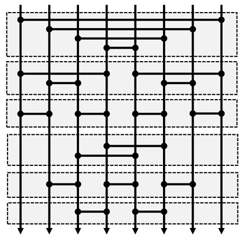
Fig. 1. A known optimal 8-way sorting network

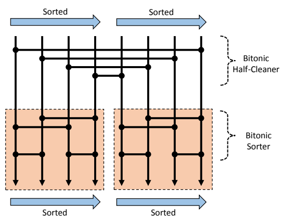

Fig. 2. Bitonic sorting network

With the data explosion from our era of "Big Data", it is not uncommon to see the need to sort dataset at hundreds-of-terabyte to petabyte scale. The de-facto solution for sorting a large-scale dataset is running MapReduce [17] programs on a cloud of general-purpose servers. To handle the complexity of the sorting job on such a large scale, a general-purpose distributed software environment, such as Apache Spark [18] and Hadoop [19], is typically deployed to manage cluster resources and schedule parallel executions. Work in this direction [20]–[23] shows that a 100TB of data can be sorted in hundreds of seconds on cluster of general-purpose servers. TecentSort [20] holds the world record of sorting 100TB of data in 98.8 seconds, which runs on top of 512 OpenPower computational nodes, connected via 100Gbps 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 [24]. Furthermore, modern processors provide Single Instruction Multiple Data (SIMD) features to more aggressively exploit data-level parallelism in sorting [1], [25]. For example, Intel processors provide SIMD features with Intel's Streaming SIMD Extensions (SSE). Work in this direction has attempted to efficiently map sorting algorithms into SSE-enabled processors [1], [2]. There is also research to accelerate sorting with larger-scale SIMD appliances such as GPGPUs [3], [4]. SIMD-enabled processors can deliver strong sorting performance on small-scale data, however such solutions are generally power-hungry.

Hardware accelerators, such as FPGAs [5]–[10] and ASICs [11]–[13], are also under active investigation to provide power-efficient solutions to sorting. Compared to SIMD-enabled general-purpose processors, hardware accelerators can deliver similar sorting performance at more than 10x lower power budget [7]. Hardware sorting accelerators can be categorized into sequential sorters and parallel sorters. Sequential sorters [9], [11] are implementations of single-threaded sorting algorithms and can produce 1 number/cycle. On the other hand, hardware parallel sorters [5], [7], [10] often implement Bitonic or odd-even sorting networks [26] 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 [13]. 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 [5].

In order to reap the maximum processing power of general-purpose or application-specific sorting engines, it is imperative that all the data should fit into the main memory of the machine. For example, if the sorting job does not fit into the aggregate DRAM capacity of a MapReduce cluster, the dataset has to be moved to the slower secondary storage, and the performance can degrade dramatically [27]–[29]. In such a setting, NAND flash memory can offer an alternative to the distributed DRAM solution for sorting [27]–[31], thanks to its orders-of-magnitude larger capacity than DRAM [32] and relatively fast random accesses compared to magnetic disks. NAND flash memory has very different characteristics than DRAM, such as accesses at page granularity and erase-before-write requirement. To migrate memory-intensive program, such as sorting, into flash storage, flash-oriented optimizations have to be made to mitigate storage overhead and reap maximum device performance [27]. Another benefit of using more flash and less DRAM is lower energy consumed by sorting tasks. NTOSort [33] holds the world record for JouleSort which measures the amount of energy to sort 1TB of data. NTOSort uses a desktop system with 16 Samsung 840 Pro 256GB SSDs, 1 Samsung 840 Pro 128GB SSD, and can sort 59,444 records/Joule.

B. Sorting Networks

1) 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. Figure 1 shows the known optimal sorting network for 8 values. 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 [34]. Figure 1 is the known optimal sorting network for input size of 8, and consists of six stages of comparators.

2) 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 part 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 2. It can be seen that it requires much fewer comparators and less depth compared to the network in Figure 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.

C. 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 [35]. Initially the main benchmark of interest was the TeraByte Sort, which measures the time to sort 1TB (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.

III. System Architecture

Figure 3 shows the overall architecture of our system. At a high level, it is simply a flash storage device with FPGA-based, in-storage accelerators. In the first step of the sorting process which is shown in Figure 4, data to be sorted is loaded onto DRAM from flash in 512MB chunks, sorted, and then written back to flash. (512MB is half the size of the on-board DRAM capacity). Three different types of sorters are involved in sorting 512MB chunks in memory: the Tuple

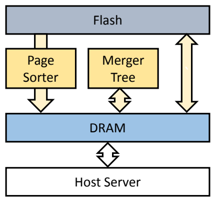

Fig. 3. The system is designed as a flash-based storage device, with FPGA-based accelerators and a DRAM buffer

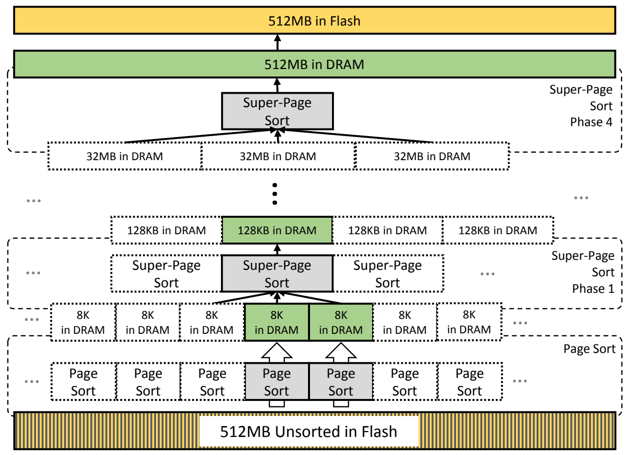

Fig. 4. Data is first sorted into chunks that can fit in the DRAM buffer and stored back in flash

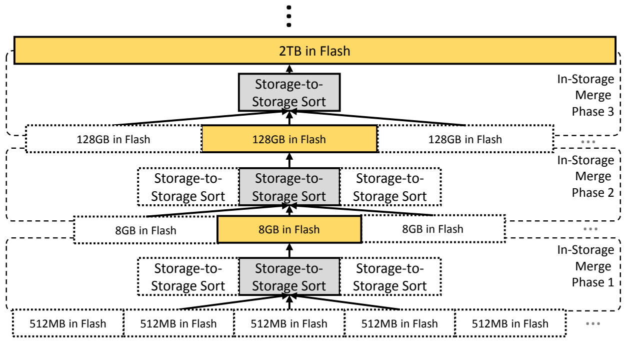

Fig. 5. Large sorted chunks are merge-sorted directly from storage to storage

sorter, Page sorter and the Super-Page sorter. Once data in flash is organized into 512MB 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 5.

A. *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 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.

B. *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 6.

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 half cleaner. When sorting in the ascending order, the values in the lower half of the result forms 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 are 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 FIFO that will be used depend on which of the two n-tuples processed in the previous cycle had the largest 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.

C. *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 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 it completely. The tuples are sorted by a sorting network before they are entered into the page sorter.

The page sorter is used to sort data into page-sized sorted chunks as it is being initially read from flash to DRAM. 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 8. Since the page sorter requires multiple passes over the data to sort it completely, multiple instance of page sorters are required to keep up with the bandwidth of the flash storage.

D. *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 as input 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

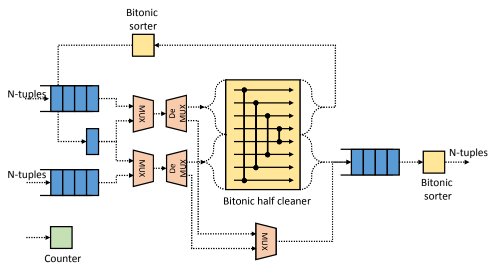

Fig. 6. Merger internal architecture

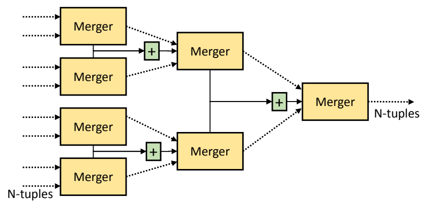

Fig. 7. A 4-leaf merge tree with 8 n-tuple input

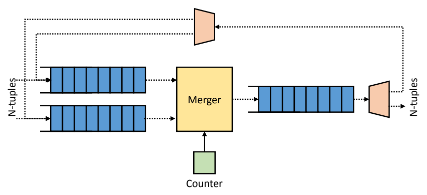

Fig. 8. Internal structure of a page sorter

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.

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 allows it. The Super-Page Sorter makes multiple passes over data stored in DRAM 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 it 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 a few KBs in size, and there is some overhead whenever a new bank has to be opened. The read performance of our 1GB SODIMM DRAM card with random 8KB page-granularity reads was about 10GB/s, while random 64-byte cache granularity reads performed at about 1GB/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 is 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.

E. Storage-to-Storage Sorter

Once data is organized into large sorted block 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 is 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.

F. Software Manager

The FPGA accelerators and the layout of data on flash is managed by a accelerator-aware file system. The file system communicates to the storage device over PCIe, and maintains a list of files in the file system and their mappings to the flash chips. It has a separate data and command paths to and from the storage device and accelerators. The data path between the storage and software operates using fast DMA communication over PCIe. The command path operates over low latency I/O mapped communication over PCIe, and is used also for sending commands to accelerators and receiving acknowledgements.

IV. Evaluation

In this section, we first describe the implementation details of the prototype system we have constructed, evaluate the performance of individual components of the system, and then explore in detail the performance characteristics of sorting a large dataset.

A. Implementation Details

We have implemented our solution using a Xilinx VC707 FPGA development board coupled with a custom flash expansion card connected via the two FMC ports on the VC707 board. The VC707 board is equipped with a Xilinx Vertex 7 FPGA and 1GB of DDR3 DRAM. The DRAM performed at 10GB/s with sequential read/writes and 1GB/s with random read/writes. The flash expansion card has a capacity of 1TB, and has a modest performance of 2.4GB/s reads and 2GB/s writes. The coupled device is plugged into the Xeon server which acts as the host, via a x8 Gen2 PCIe slot. FPGA development was largely done in the Bluespec hardware description language.

B. Component Performance

1) Merger Tree: The merger tree was run at a clock frequency of 125MHz, using a data path of 256 bits. We measured the performance of the merger using values sizes of of 64 bits and 128 bits. At a data path of 256 bits, the merger

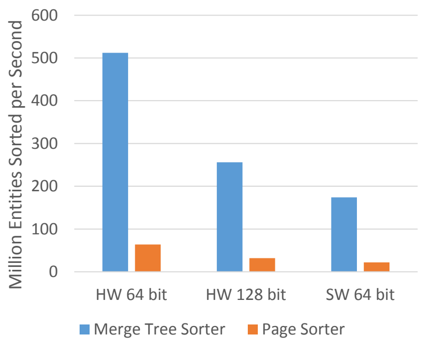

Fig. 9. Throughput of Merge Tree and Page Sorter for 64 and 128 bit hardware implementation compared to a 64 bit software implementation

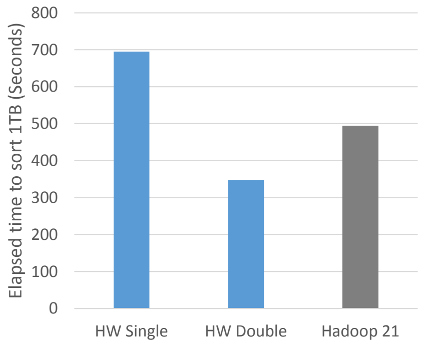

Fig. 10. Single node performance exceeds a 21-node Hadoop cluster with two accelerated storage devices

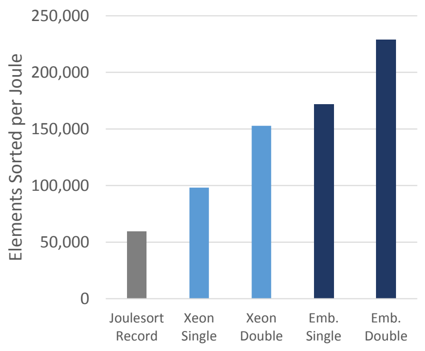

Fig. 11. Prototype implementation achieves almost twice the power-performance compared to current Joulesort record. Better implementation is expected to achieve 4 times the power-performance.

takes as input 4-tuples and 2-tuples, respectively. The merger invariably emits one n-tuple at every cycle regardless of data distribution. At 125MHz, it merges data at 4GB/s, regardless of value size.

2) Page Sorter: The page sorter sorts 8KB 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 0.5GB/s. *Since the goal of the page sorter is to sort data as it is read from flash, our system required 5 page sorters to completely saturate the 2.4GB/s flash bandwidth.*

Figure 9 compares the performance of merge trees and page sorters implemented for hardware of various value sizes, and a single thread software implementation. The software implementation was compiled with GCC with -O3 optimizations and run on an Intel Xeon E5-2690 running at 2.90GHz.

3) External Sorters: A 16-way merger is capable of emitting a sorted tuple at every cycle, resulting in a 4GB/s bandwidth per merger. There are three different situations for the use of 16-way merge-sorters: *Sorting from DRAM to DRAM, which requires two 16-way mergers to saturate the bandwidth, DRAM to flash and flash to flash, which require one merger.* The maximum available bandwidth in each situation is 5GB/s, 2GB/s and 1GB/s, respectively.

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

C. End-to-End Sorting Performance

We demonstrate the end-to-end performance of the system by sorting 512GB of data stored in flash. We experimented with 512GB of data because the effective capacity of a 1TB flash storage is actually a bit smaller due to increasing number of bad blocks due to use, and some additional storage capacity for intermediate buffers are required by the system.

Sorting is done in phases as is traditional merge sort. At the first phase, unsorted data is read in 8KB blocks and sorted via the page sorter, after which the sorted block size is 8KB. At every subsequent sort phase, all blocks are merged 16 blocks at a time through a merge tree, increasing the sorted block size by 16-fold after every phase.

Table II shows the breakdown of performance across the merge phases to completely sort 512GB of data, along with the maximum read bandwidth each medium can provide during that phase. Each row corresponds to a merge sort phase. The elapsed time column shows the total amount of time it took to merge all sorted blocks from the previous phase. All elapsed time values were averaged over multiple executions and rounded up to the nearest 10 seconds. We can see that at every stage, almost the maximum bandwidth of the mediums is being saturated for useful work. The numbers we are getting is the maximum sorting performance available from these storage hardware.

An interesting effect of using high fan-out mergers is that as long as the number of merge phases do not change, the amount of time it takes to sort a dataset is linear to its size. The range of datasets with the same number of merge phases that include 512GB is over 128GB to 2TB. For example, 256GB can be sorted in 1200 seconds, and 2TB can be sorted in 10,000 seconds.

Sorted Block Size log(bytes)	Medium	Bandwidth (GB/s)	Time (s)
13	flash-DRAM	2.4	220
17	DRAM-DRAM	5	110
21	DRAM-DRAM	5	110
25	DRAM-DRAM	5	110
29	DRAM-flash	2	280
33	flash-flash	1	520
37	flash-flash	1	520
41	flash-flash	1	520
		total	2390

TABLE II
512GB SORTING PERFORMANCE BREAKDOWN BY MERGE PHASE

D. 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 1TB of such key-value pairs, and stored the key and value data separately. The keys were augmented with a pointer to its corresponding value. As a result, the actual dataset to be sorted was a list of 16-byte key-pointer pairs. It takes our system 700 seconds to completely sort 10^{10} 16-byte key-pointer pairs, or 150GB of data.

Published performance numbers on Terasort from MapR reports 494 seconds to completely sort 1TB of data, or 10^{10} keys, on a 21 node cluster, each node equipped with 32 cores, 128GB of RAM and 11 HDDs [15]. With a single storage device, our system performs at more than half the performance of a 21-node MapR cluster. In order to exceed the cluster performance, we can simply add another accelerated storage device and double the available bandwidth. This allows us to completely sort the data in less than 400 seconds with a single node. The comparison between accelerated systems with one or two storage devices and the Hadoop system can be seen in Figure 10.

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.

E. Power-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 actually 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 140W of power, of which a single accelerated storage device is responsible for about 40W. 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 11 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 40W of power. Terasort benchmark data is packed into 128 bits of key-pointer pairs. The software numbers are from NTOSort [33], which holds the current record for Joulesort. Although our custom hardware violates the constraints for the Joulesort benchmark, it is a good reference point to compare against. Not only does our prototype outperform the current record holder by almost twice the efficiency, a projected system with more realistic components for deployment outperforms the current record holder by almost four times.

V. CONCLUSION

In this paper, we have 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 exceed the power efficiency of the current Joulesort record holder by up to four times. The device components we used to construct our prototype are not special among their peers. For example, a 1TB PCIe SSD with 2.4GB/s of bandwidth is not a particularly high performance device anymore. We think a system built using our design, using more modern and less wasteful components can further improve the performance and power efficiency of sorting systems.

It should be noted that because Page Sorters and In-Memory Sorters remove fine-grained random access into storage, various storage characteristics such as read granularity or random access performance becomes unimportant. As a result, our system can perform well using other storage devices, such as HDDs or other future NVMs, as long as it delivers high sequential throughput.

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. We plan to continue exploring the use of flash and FPGA accelerators to enable more low-power high-performance analytics of more complex problems.

REFERENCES

[1] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1313–1324, Aug. 2008.
[2] H. Inoue and K. Taura, "SIMD- and Cache-friendly Algorithm for Sorting an Array of Structures," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1274–1285, Jul. 2015.
[3] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–10.

[4] N. Govindaraju, J. Gray, R. Kumar, and D. 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*. New York, NY, USA: ACM, 2006, pp. 325–336.

[5] 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)*, May 2016, pp. 95–102.

[6] 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)*, June 2011, pp. 38–45.

[7] R. Mueller, J. Teubner, and G. Alonso, "Sorting Networks on FPGAs," *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, Feb. 2012.

[8] 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)*, Dec 2009, pp. 431–434.

[9] D. Koch and J. 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*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 45–54.

[10] J. Casper and K. Olukotun, "Hardware Acceleration of Database Operations," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 151–160.

[11] K. ystein Arisland, A. C. Aasb, and A. Nundal, "VLSI parallel shift sort algorithm and design," *Integration, the VLSI Journal*, vol. 2, no. 4, pp. 331 – 347, 1984.

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

[13] A. Farmahini-Farahani, H. J. D. III, M. J. Schulte, and K. Compton, "Modular Design of High-Throughput, Low-Latency Sorting Units," *IEEE Transactions on Computers*, vol. 62, no. 7, pp. 1389–1402, July 2013.

[14] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[15] *TeraSort Benchmark Comparison for YARN*, 2014 (Accessed January 18, 2017). [Online]. Available: https://www.mapr.com/sites/default/files/terasort-comparison-yarn.pdf

[16] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "Joulesort: a balanced energy-efficiency benchmark," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 365–376.

[17] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, 2010, p. 10.

[19] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[20] J. Jiang, L. Zheng, J. Pu, X. Cheng, C. Zhao, M. R. Nutter, and J. D. Schaub, *Tecent Sort*, 2016 (Accessed January 11, 2017). [Online]. Available: http://sortbenchmark.org/TencentSort2016.pdf

[21] J. Wang, Y. Wu, H. Cai, Z. Tang, Z. Lv, B. Lu, Y. Tao, C. Li, J. Zhou, and H. Tang, *FuxiSort*, 2015 (Accessed January 11, 2017). [Online]. Available: http://sortbenchmark.org/FuxiSort2015.pdf

[22] R. Xin, P. Deyhim, A. Ghodsi, X. Meng, and M. Zaharia, *GraySort on Apache Spark by Databricks*, 2014 (Accessed January 11, 2017). [Online]. Available: http://sortbenchmark.org/ApacheSpark2014.pdf

[23] T. Graves, *GraySort and MinuteSort at Yahoo on Hadoop 0.23*, 2013 (Accessed January 11, 2017). [Online]. Available: http://sortbenchmark.org/Yahoo2013Sort.pdf

[24] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, "Merge Path - Parallel Merging Made Simple," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1611–1618.

[25] H. Inoue, T. Moriyama, H. Komatsu, and T. 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*, ser. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 189–198.

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

[27] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks, and Arvind, "BlueCache: A Scalable Distributed Flash-based Key-value Store," *Proc. VLDB Endow.*, vol. 10, no. 4, pp. 301–312, Nov. 2016.

[28] S.-W. Jun, M. Liu, K. E. Fleming, and Arvind, "Scalable Multi-access Flash Store for Big Data Analytics," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 55–64.

[29] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: An appliance for big data analytics," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 1–13.

[30] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1075–1086.

[31] S.-H. Kang, D.-H. Koo, W.-H. Kang, and S.-W. Lee, "A Case for Flash Memory SSD in Hadoop Applications," *International Journal of Control and Automation*, vol. 6, no. 1, 2013.

[32] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70.

[33] A. Ebert, *NTOSort*, 2013 (Accessed January 11, 2017). [Online]. Available: http://sortbenchmark.org/NTOSort2013.pdf

[34] D. E. Knuth, *The art of computer programming: sorting and searching*. Pearson Education, 1998, vol. 3.

[35] J. Gray, C. Nyberg, M. Shah, and N. Govindaraju, *Sort Benchmark Home Page*, 2016 (Accessed January 11, 2017). [Online]. Available: http://sortbenchmark.org/