Minerva: Accelerating Data Analysis in Next-Generation SSDs

Arup De*† Maya Gokhale† Rajesh Gupta* Steven Swanson*
*Department of Computer Science and Engineering
University of California, San Diego
{arde,rgupta,swanson}@eng.ucsd.edu
†Lawrence Livermore National Laboratory
gokhale2@llnl.gov

***Abstract*—Emerging non-volatile memory (NVM) technologies have DRAM-like latency with storage-like density, offering unique capability to analyze large data sets significantly faster than flash or disk storage. However, the hybrid nature of these NVM technologies such as phase-change memory (PCM) make it difficult to use them to best advantage in the memory-storage hierarchy. These NVMs lack the fast write latency required of DRAM and are thus not suitable as DRAM equivalent on the memory bus, yet their low latency even in random access patterns is not easily exploited over an I/O bus.**

In this work, we describe an FPGA-based system to execute application-specific operations in the NVM controller and evaluate its performance on two microbenchmarks and a key-value store. Our system *Minerva*[1] extends the conventional solid-state drive (SSD) architecture to offload data or I/O intensive application code to the SSD to exploit the low latency and high internal bandwidth of NVMs. Performing computation in the FPGA-based NVM storage controller significantly reduces data traffic between the host and storage and serves as an offload engine for data analysis workloads. A runtime library enables the programmer to offload computations to the SSD without dealing with the complications of the underlying architecture and inter-controller communication management. We have implemented a prototype of Minerva on the BEE3 FPGA system. We compare the performance of Minerva to a state of the art PCIe-attached PCM-based SSD. Minerva improves performance by an order of magnitude on two microbenchmarks. Minerva based key-value store performs up to 5.2 M *get* operations/s and 4.0 M *set* operations/s which is 7.45× and 9.85× higher than the PCM-based SSD that uses the conventional I/O architecture. This huge improvement comes from the reduction of data transfer between the storage to the host and the FPGA-based data processing in the SSD.

***Index Terms*—storage systems, non-volatile memory, solid-state drive, I/O performance, flash memory, phase-change memory, spin-transfer torque memory, FPGA, Virtex-5 and big data applications.**

I. Introduction

In the age of Big Data, data analysis workloads are increasingly affected by bottlenecks in the memory-storage hierarchy. Enterprise and scientific data analysis applications employ complex algorithms that require efficient techniques, tools and infrastructure to better interact with petabytes of data. For example, semantic graphs representing social networks of interest to the Department of Homeland Security will have 10^{15} entities [18]. Execution time of such data analysis algorithms is dominated by access time to disk and flash. Emerging fast, byte-addressable non-volatile memory (NVM) technologies, such as phase-change memory (PCM) [5] approach DRAM-like performance with the additional benefits of lower power consumption and higher density as process technology scales. Despite greatly improved latency compared to disk, the asymmetric read/write latencies and wear characteristics of these devices make them less suitable for memory bus attachment than I/O bus (e.g., PCIe, SATA etc.). Also, PCIe facilitates better scalability as compared to parallel memory bus interface. However, existing block-based storage I/O interfaces cannot fully utilize these random-access memories, and become potential bottlenecks for future computing systems using NVM. To eliminate I/O bottlenecks and better utilize NVM capabilities, we propose a novel FPGA-based SSD architecture, called *Minerva*. Minerva moves computation close to data to greatly reduce the amount of data that must travel across a slow I/O interface (e.g. PCIe) through main memory and caches to reach the compute units. Creating a capability to compute in FPGA storage controllers is particularly timely as commodity SSD controllers are often implemented on FPGA and the near-DRAM read latency of PCM provide a compelling incentive to place computation close to the storage devices.

In this paper we describe the design and implementation of the Minerva compute-capable NVM storage controller. Our controller extends one of the first PCM storage array controllers Moneta-D[7], by incorporating a storage processor module directly connected to each memory controller. We describe the hardware infrastructure to schedule and dispatch compute kernels. Our design uses a generic command/status mechanism to push computation to storage controllers and retrieve results. A runtime library enables the programmer to offload computations to the SSD without dealing with many of the complications of the underlying architecture and inter-controller communication management.

Minerva is suitable for large scale applications that rather than reusing data, stream over large data structures or randomly access different locations. As a result, they make poor use of existing memory hierarchies and perform poorly on the conventional system due to a large I/O overhead.

We have built a prototype of Minerva on the BEE3 FPGA prototyping system [4] and evaluate the performance of Minerva on two microbenchmarks and a key-value store.

[1]"Minerva" is the Roman goddess of intelligence.

TABLE I
MEMORY TECHNOLOGY SUMMARY [16], [19], [22]

Technology	Density	Latency		Energy		Idle Power/GB
		Read	Write	Read	Write	
Flash	4 F^2	25 us	200 us	250 pJ/bit	250 pJ/bit	10 mW
PCM	4 F^2	67.5 ns	215 ns	3.4 pJ/bit	17.84 pJ/bit	1 mW
DRAM	4 F^2	25 ns	25 ns	2.4 pJ/bit	2.4 pJ/bit	100 mW

We compare the performance with Moneta-D and find that Minerva outperforms Moneta-D by an order of magnitude on two microbenchmarks. A Minerva based key-value store performs up to 5.2 M *get* operations/s and 4.0 M *set* operations/s which is 7.45× and 9.85× higher than Moneta-D. This huge improvement comes from the reduction of data transfer from the storage to the host, the elimination of I/O, and the efficient data processing in the SSD.

The rest of the paper is organized as follows. Section II presents emerging NVM technologies and motivations. Section III describes the Minerva architecture, the programming model and implementation. Section IV presents the results of our experiments. Section V discusses related work and Section VI concludes the paper.

II. BACKGROUND AND MOTIVATION

In the last five decades, the disk has been the basic unit of persistent storage for small and large scale computing systems. Disks are highly reliable but generally suffer from poor latency and bandwidth, especially relative to processor and main memory advances. Recently, flash based PCIe-attached SSDs have become popular (e.g. FusionIO [14] and Virident [26] as representative of the high end SSDs) and are significantly faster (100×) than disk but still have slow erase, and high write latency. Emerging NVM technologies such as phase change memory (PCM) and spin-transfer torque (STT) are narrowing the performance gap between the storage and the main memory and have potential, with density improvements, to replace existing storage technologies in future. Table I briefly summarizes density, latency and energy dissipation of different memory technologies.

Phase change memory (PCM) is the most promising of the upcoming NVM technologies [5]. It exploits the property of chalcogenide glass to switch between two states, amorphous (high-resistance) and crystalline (low-resistance), with application of current pulses. The crystalline state achieves by heating above crystallization temperature using a moderate, long current pulse, and logically stores “1”. The amorphous state achieves by high, short current pulse and logically stores “0”. A small read current (less than 100uA) used to sense data stored in a cell by measuring its resistance thus PCM consumes very small power for read. PCM approaches DRAM-like performance with lower power consumption and higher density as process technology scales. Despite this promise, PCM suffers from long write latency, high energy writes and limited write endurance (on the order of 10^8). Recent studies [19], [24], [23], [13] proposed several hardware and software enhancements such as various wear-leveling methods, new row buffer design, selective writes, (DRAM/PCM) hybrid memory architecture, hot page swapping and write buffers to overcome PCM technology

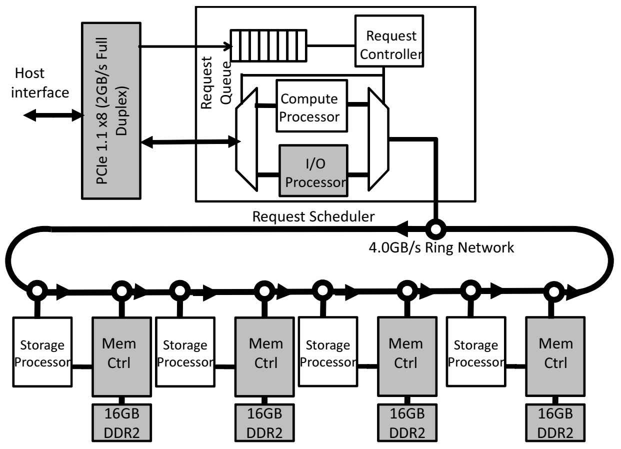

Fig. 1. **The Minerva architecture** builds on Moneta-D (gray boxes) with an augmented scheduler and storage processors(white boxes).

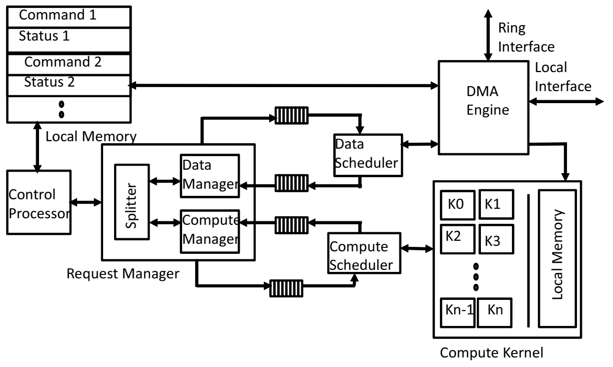

Fig. 2. **The storage processor components** are responsible for running application code in the storage array. Each storage processor connects to a local memory controller to operate on locally stored data. It also connected with a ring network to communicate with other components.

limitations and make them a viable option for charge-based memory replacement such as DRAM and flash. The Minerva architecture is motivated by the desire to retain the conventional I/O bus storage architecture while simultaneously exploiting the low access latency (300× faster than flash) and byte addressability of PCM within the storage array by performing application-specific computation in storage.

The Moneta project [6] designed and implemented an SSD storage controller optimized for small, random access to byte addressable PCM. The controller was implemented on the BEE3 prototyping platform with emulated PCM latencies to access data in the 64 GB DRAM. Moneta was further optimized to perform fast file system operations such as metadata access and update, the Moneta-D [7], which is the base system upon which Minerva is implemented. the Onyx [3] storage array replaces the DRAM in the BEE3 with first generation PCM DIMMs.

III. MINERVA

This section describes Minerva’s architecture, programming model and prototype implementation.

A. Minerva Architecture

The Minerva architecture improves performance by offloading application-specific computations to the storage

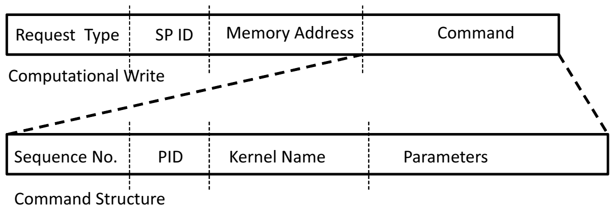

Fig. 3. **The computational write request** extends the semantics of memory write to initiate application specific commands in the storage processor.

array. It exposes the high-bandwidth and low-latency NVM accesses to user applications and reduces data traffic across the I/O interface. Figure 1 shows the architecture of the Minerva. We extended Moneta-D [7] (gray boxes in Figure 1) with a request scheduler and storage processors (white boxes in Figure 1). The PCIe interface and request scheduler reside on FPGA 0 of the 4-FPGA ring-connected BEE3. Each FPGA also holds a memory controller and associated storage processor. Minerva has four major components: request scheduler, storage processor, network and memory controller. The request scheduler receives requests from the host using a Programed IO (PIO) via the PCIe channel and places them into a FIFO queue. The host issues a computational write request to trigger application-specific processing on a storage processor and a computational read request to receive a response from the storage processor. Normal read and write I/O requests are passed to the I/O processor. Each memory controller connects with dual-rank DIMMs using the DDR2 interface. The network provides a packet based communication among the request scheduler, storage processors and memory controllers, and has bandwidth and round trip latency of 4 GB/s and 88 ns respectively. We describe each component in more detail below.

Request scheduler The request scheduler receives an I/O or computational request from the host. Each request has an additional field called *type* to distinguish between conventional I/O or computational request. The request scheduler has three main components: request controller, I/O processor and compute processor. The request controller retrieves each request from the FIFO, checks the opcode and routes requests between the compute processor and the I/O processor based on opcode and manages shared resources. The I/O processor handles the conventional read and write I/O requests. The compute processor handles computational requests, which specify the storage processor (SP) ID and the memory address information to dispatch computations to the SP. Although the emulated PCM is divided into four 16 GB groups, there is a global 64 GB address space, and file data is striped across the four memory banks in 8 KB slices.

Figure 3 shows the *computational write* request. It has following fields.

- Request Type - identifies the type of request such as read or write
- SP ID - identifies the storage processor
- Memory Address - identifies the command location
- Command - holds application related information.

When the host sends a computational write request to start computation in the storage, the compute processor issues a DMA command to transfer request data from the host's memory into its buffer. When the DMA transfer completes, the compute processor checks the status of the target SP. If the SP has adequate space to receive request, the compute processor dispatches the request via the ring network; otherwise it waits until sufficient space is available. Once the scheduler gets acknowledgment from the SP, it raises an interrupt and sets a tag status bit. The operating system receives the interrupt and completes the request by reading and then clearing the status bit using PIO operation. The *computational read* requests are handled similarly but in the reverse direction.

Storage Processor The storage processor (SP), shown in Figure 2, acts as a coprocessor to the host. The storage processor communicates with the local memory controller using the local interface (bandwidth 4 GB/s). It communicates with the request scheduler and other memory controllers and storage processors using ring interface (bandwidth 4 GB/s).

The SP consists of control processor, request manager, DMA engine, data scheduler, compute scheduler, and compute kernel. Each host process that wants to issue computation requests in the storage processor is given a command and status area in the control processor's local memory which is setup, managed and released by the driver.

The host dispatches computations by writing in the command area associated with that process. The host can continue executing asynchronously or can wait for the completion of command processing by polling for status. When instructions exist in command area, the control processor fetches and executes instructions from the command buffer.

The control processor is an in-order RISC processor which issues DMA requests to load or store file data from or to memory controllers and dispatches application-specific computations to compute kernels via the request manager. The control processor also performs scalar operations on behalf of the application.

The request manager receives DMA or computational requests from the control processor, breaks large requests to multiple small requests and places them into a queue of the data scheduler or compute scheduler respectively. The DMA engine efficiently loads data from the memory controller to local memory based on the request from the data scheduler and vice-versa. The data scheduler issues multiple DMA requests, maintains a scoreboard and sends back acknowledgment to the request manger on completion. The compute kernel has application specific code for hardware acceleration and operates on data in local memory based on computational requests from the compute scheduler. The compute scheduler triggers application-specific kernels, checks status and returns result to the request manager on completion.

The compute kernel is an application-specific hardware module that receives commands from the compute scheduler. Each kernel processes data in local on-chip RAM. We have developed several custom kernels manually in Verilog for evaluation (see Section IV). Custom kernels offer the highest

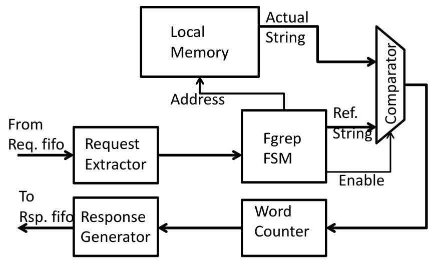

Fig. 4. **The fgrep-8 kernel** receives compute requests, finds matching strings, updates the word match counter, and sends a response to the compute scheduler.

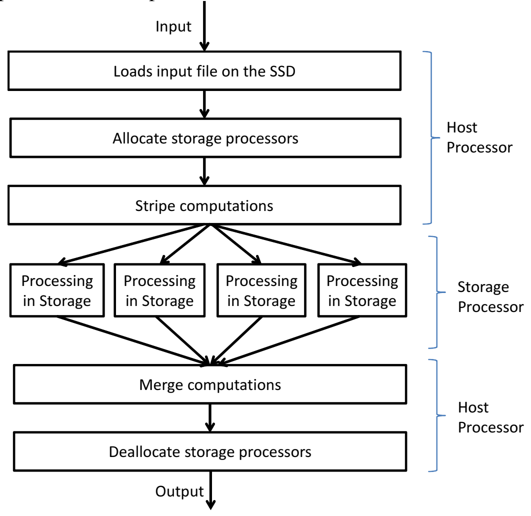

Fig. 5. **The execution flow on Minerva based system** is initiated by the host, runs asynchronously on the storage processor, and is terminated by the host.

performance at the corresponding cost of hardware design expertise and the requirement to interface to the compute scheduler.

As an example, Figure 4 shows a simple streaming kernel, fgrep-8, which receives compute requests from the compute scheduler. The request extractor extracts the reference string and data location information. The data may reside in a local memory, in which case it is loaded by the local controller, or may reside in a remote memory. In the latter case, the data must be read by the remote memory controller and transferred via the ring network to a memory block local to the requesting processor. The fgrep-8 FSM compares the reference pattern with the data in the local memory block. On matches, the kernel increments the match word counter and when all the requested data has been searched, it sends back the results to the compute scheduler via the response generator. The control processor updates the status block based on the response of command processing and generates an interrupt for completion notification. It also generates an interrupt in case of failure or situations that require host intervention to resolve. For example, the storage processor does not initiate I/O operations for safety and simplicity. The host issues I/O operations upon request from the storage processor.

```
int64_t GrepPthread(int fd, int64_t offset,
        size_t size, int64_t key)
{
 // Allocate local buffer
 buf = (int64_t*) malloc(buf_size);
 // Read file with given offset
 pread (fd, buf, buf_size, offset);
 // Search for key and increment
 for(i = 0, i < buf_size, i++){
  if(buf[i] == key)match_count += 1;
 }
 return match_count;
}
```

Fig. 6. The conventional fgrep-8 code snippet

B. Minerva Programming Model

We propose a generic command/status mechanism to dispatch computations which allows applications to be executed efficiently and safely. The command/status API is managed by a software driver, which is responsible for setup, management, monitoring and release of the SP on behalf of a given application process.

The application uses the command data structure to offload application-specific processing to the storage processor and uses status data structure to read the response from the SP. The command structure contains command sequence number, PID, application's kernel name and associated parameters such as data and file extents. The command sequence number is incremented once for each request and provides a unique command identifier. The storage processor processes user commands and updates the status structure. The status structure comprises of a unique command sequence number, command status and results. The command status field indicates the completion of a given command.

We provide the write function `sppwrite` to send commands to the storage processor from the host. This is analogous to existing file write function `pwrite` but takes the storage processor's ID and command location instead of file descriptor and file offset as input arguments. Similarly, we read status from the storage processor using `sppread`.

Figure 5 shows the execution flow of an application using the Minerva architecture. If needed, the host processor loads file data onto the SSD. Files are striped across multiple memory controllers for parallelism. Next, the host processor allocates a particular command and status structure for that application and then the host distributes computations across multiple storage processors through the command data structure. The storage processors process application-specific code and update their status data structures. During computation, if a storage processor is asked to access data outside its local stripe(s), it fetches that data from remote memory controllers. When the command has been completed, the host merges results from different storage processors and produces the output. Finally, the host processor deallocates the command and status structures associated with that application.

As an example, the fgrep-8 benchmark scans through a data file, searches for a specified 8-byte character pattern and counts the number of occurrences. Figure 6 presents a code snippet illustrating a parallel fgrep-8 function using

```
int64_t GrepMinervaHost(int fd, int64_t offset,
        size_t size, int64_t key)
{
 int64_t match_count = 0;
 // Create command: which consists
 //kernel name, extents and key
 cmd_buf = CreateCmd("fgrep",
   fd, offset, size, key);
 // Write the command
 sppwrite (sp_id, cmd_buf, buf_size, cmd_offset);
 // Check for completion notification
 // and then read status
 while(!complete);
 spppread (sp_id, status_buf, buf_size,
   status_offset);
 // Read match count from status buffer
 match_count = get_match_count(status_buf);
 return match_count;
}
```

Fig. 7. Minerva's fgrep-8 host code snippet

```
void GrepMinervaSP(char* command, char* status)
{
 int64_t local_match_count = 0;
 // Retrieves kernel name,
 // extents and key from the command
 kernel = GetKernelName(command);
 extents = GetExtents(command);
 key = GetKey(command);
 // Read data from local memory
 // controller and checks key
 for(i = 0;i < extents_count;i++){
   DmaData(extents, buf);
   match_count +=
     RunKernel(kernel, buf, key);
 }
 // Update status and send
 // completion notification
 UpdateStatus(status, match_count);
}
```

Fig. 8. Minerva's fgrep-8 storage processor kernel snippet

pthreads. The data file is split across multiple threads and each thread reads the file at its assigned offset. The thread reads "size" bytes of data from the storage array into a local buffer. Then it searches for a specified character pattern and increments *match_count* when a match found. Finally, the main program combines all the *match_count*s to get the total number of matches.

The Minerva version has two parts: the host application code and the fgrep-8 kernel code on the storage processor. The host application initiates computation using a computational write command. The command comprises of the kernel name and input arguments such as file extents and key along with other control information. Our run-time library function extracts information on the data file extents from file system. The host application sends the command using `sppwrite` and waits for the storage processor's notification on completion. The storage processor retrieves the kernel name, file extents and key from the command. Then it reads file data based on the extents information using DMA, and counts the number of matches using the fgrep-8 kernel (Figure 4). Finally, it updates status field with the match count and issues completion notification. When host receives completion notification it reads status and gets the *match_count*. Figure 7 and Figure 8 present the Minerva's fgrep-8 code snippet of the host and storage processor respectively. As an optimization, we also implement *double buffering* to simultaneously load data from the memory controller to one local buffer and execute the application-specific kernel on another buffer.

TABLE II
RESOURCE USAGE ON VIRTEX-5 LX155T

Component	Slice Regs		LUTs		BRAMs	
		%		%		%
Request Scheduler	10536	10.8	11509	11.8	60	28.3
Storage Processor	18359	18.8	14408	14.8	24	11.3
Memory Controller	6779	6.9	5405	5.5	18	8.4
PCIe-1.1 x8	3882	3.9	3008	3.0	11	5.1
Ring Network	984	1.0	856	0.8		

C. Minerva prototype

We implemented Minerva on the BEE3 FPGA prototyping system jointly developed by Microsoft Research, UC Berkeley, and BEEcube Inc.. The BEE3 system holds 64 GB of 667 MHz DDR2 DRAM under the control of four Xilinx Virtex-5 LX155T FPGAs, and it provides a PCIe-1.1 x8 (2 GB/s full duplex) link to the host system. We used the high speed DDR2 ring network to connect multiple FPGAs with roundtrip latency of 88 ns and bandwidth of 4 GB/s. The system clock runs at 250 MHz. We implemented the hardware components such as request scheduler, storage processors and memory controllers on the FPGAs. Table II presents the FPGA resource usage of different components.

Since PCM is the most promising among different NVM technologies and receives great attention in research community as a viable future replacement of existing charge based memory technologies, we use PCM for our evaluation. We emulate PCM using the DRAM of the BEE3 system with access latency as described in [19]. We modify the memory controller to add latency between the read address strobe and column address strobe commands during reads and extend the pre-charge latency after a write by inserting delay. We cannot stop DRAM refresh to preserve data which is not required for these NVM technologies.

D. Minerva Latency and Bandwidth Characteristics

We measure the latency of 4 KB page accesses for a storage processor on Minerva and the host using a conventional I/O request to Moneta-D [7]. The storage processor's access time is significantly less than the host's as the request and data don't travel over the PCIe bus and don't incur software driver overhead. The Minerva storage processor sees a latency of 1.5 μs for local 4 KB page access which is almost 82% reduction as compared to Moneta-D. The storage processor bandwidth varies with the degree of locality. When a data set is not fully local, the storage processor must read/write data from/to other memory controllers over the 4 GB/s ring network. We introduce a new term called *remote access rate* which is the ratio of number of remote memory controller accesses to total number of accesses. Figure 9 shows the bandwidth as a function of the remote access rate for Minerva and Moneta-D. The remote access rate 0.0 is the best case when a dataset is fully local

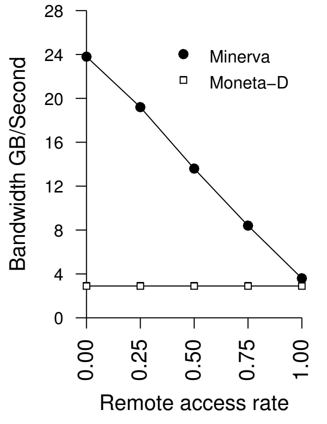

Fig. 9. **The bandwidth characteristics as a function of remote access rate for Minerva and Moneta-D [7]**

TABLE III
MICROBENCHMARKS

Name	Data Sets	Description
Fgrep-8	32 GB	Scans a file for a given 8-byte pattern matching.
RMW	32 GB	Read-modify-write random file locations.

to the storage processor and Minerva achieves aggregate bandwidth of 23.8 GB/s. Then we increase remote access rate by increasing number of accesses on remote memory controllers. We observe a linear drop in aggregate bandwidth as we increase the remote access rate. In worst case, when all accesses from the remote memory controller we achieve an aggregate bandwidth of 3.6 GB/s. The application developer can optimize performance by partitioning the data set and computation to reduce the number of remote accesses. With conventional host-based I/O using Moneta-D, the PCIe interface limits the overall bandwidth of Moneta-D and thus the abundant PCM bandwidth inside SSD remains untapped. Moneta-D achieves aggregate bandwidth of 2.9 GB/s.

IV. EVALUATION

We begin our evaluation by focusing on I/O and data intensive microbenchmarks and the impact of Minerva on overall execution time. The host machine is two-socket, Core i7 Quad machine running at 2.26 GHz with 16 GB of physical DRAM and two 8 MB L2 caches (one per socket). We compare Minerva's performance with Moneta-D [7] to demonstrate that the performance gain of Minerva is not only from using PCM but also its hardware and software enhancements. We also present sensitivity analysis results including the impact of number of memory controllers and the data size on overall performance. Then we study the impact on a key/value data store.

A. Microbenchmarks

Table III shows benchmarks for evaluation. They represent two different types of applications. Fgrep-8 has sequential accesses and the I/O interface bandwidth limits the performance of Fgrep-8. Minerva exploits massive bandwidth and parallelism of multiple memory controllers to significantly improve performance. RMW has small random accesses on different file locations. It performs poorly on the conventional architecture because of high I/O overhead. Minerva accelerates the performance by eliminating I/O and performing efficient FPGA-based data processing in the SSD. We developed a conventional parallel version of those benchmarks using pthreads for Moneta-D and then developed a Minerva version. We briefly describe the conventional and the Minerva version of each benchmark.

Fgrep-8 Fgrep-8 scans through the data file, searches for a specified 8 byte character pattern and counts the number

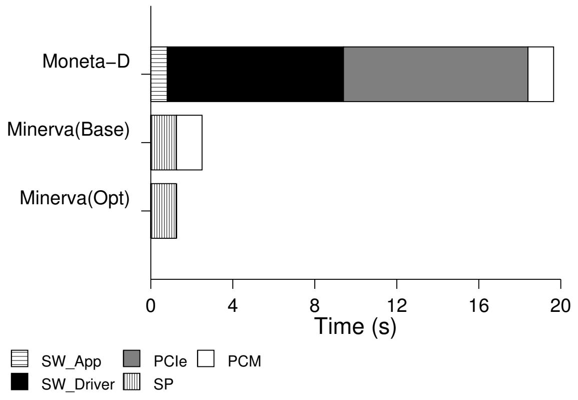

Fig. 10. **The execution time breakdown for the grep on Minerva and Moneta-D**.

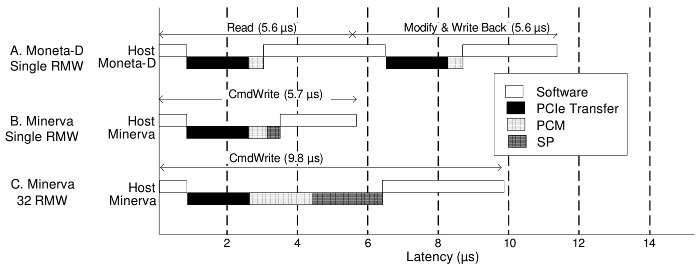

Fig. 11. **The latency comparison for RMW on Minerva and Moneta-D (Read/Write** 512 **B)**.

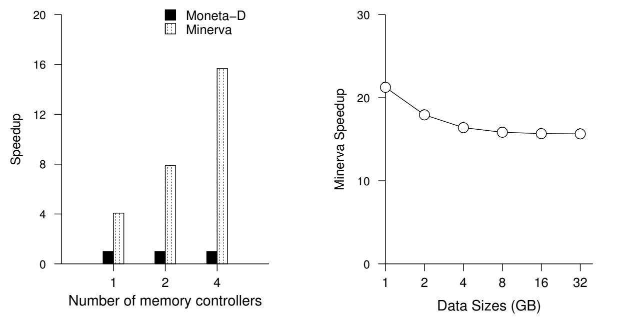

Fig. 12. **The scalability analysis** of Minerva as compared to Moneta-D.

of occurrences. We discussed its implementation detail in Subsection III-B.

RMW RMW measures read-modify-writes-per-second. It randomly reads and updates different locations of a file. In the conventional version, each update needs to read existing value from the file (read I/O), update it (increment by 1) and finally write back updated value to the file (write I/O). In Minerva version, the host sends a computational command to the storage processor with a given file location. The storage processor reads the data, modifies it and writes it back to the file, and finally sends notification to the host on completion. We further optimize it by agglomerating multiple such operations in a single computational command. We implemented this feature in our user-space library.

B. Microbenchmarks Performance Analysis

Fgrep-8 has sequential accesses and is mainly I/O bandwidth limited which indicates a large scope of improvement with Minerva. Figure 10 shows the execution time breakdown of Moneta-D and Minerva for the fgrep-8 benchmark with file size 32 GB. Moneta-D achieves maximum performance with 16 threads configuration and the total execution time is 19.65 sec. This is almost 4× speedup over single-threaded performance due to better utilization of the I/O stack and multi-core system. However, the performance of Moneta-D is mainly limited by the PCIe interface and the software driver overhead. Moneta-D achieves read I/O bandwidth of 1.6 GB/s which is 6.7% of SSD's internal bandwidth. Minerva exposes high internal bandwidth of the SSD and eliminates storage I/O overhead by offloading computations to the SSD. Fgrep-8 exploits the internal bandwidth across multiple controllers by running parallel stream kernels in each memory controller and keeping data movement local to each controller. It also uses double buffering to effectively utilize computational resources. Minerva achieves aggregate read bandwidth of 23.6 GB/s and takes total execution time 1.25 sec. It outperforms Moneta-D by 15.7×.

RMW has random accesses and mainly depends on I/O latency. Figure 11 shows the latency of Moneta-D and Minerva for RMW benchmark. For single RMW, Minerva reduces the total latency by half using one computational request. For multiple RMW, Minerva agglomerates multiple requests to one computational requests and significantly reduces the overall execution time. Minerva achieves throughput of 1.1 million ops/s which is 24× faster than Moneta-D.

Figure 12 (left) shows the speedup of Minerva with different number of memory controllers for Fgrep-8. The input data file size is fixed at 32 GB. In Moneta-D, we observe a little performance improvement when the number of memory controller(s) increases from one to two because of data striping across multiple controllers but saturate after that due to PCIe bandwidth limitations. Minerva achieves significant performance improvement and is scalable with increasing number of memory controllers. In addition, we experiment with different sizes of dataset. Figure 12 (right) shows the performance of the Fgrep-8 where we vary the data set size from 1 GB to 32 GB with fixed number of memory controllers (4) for Moneta-D and Minerva. Minerva's speedup ranges from 15.7× to 21.25×. It achieves maximum speedup for 1 GB data since Moneta-D can not exploit the full PCIe bandwidth for small datasets whereas Minerva exploits the full internal bandwidth of SSD. Eventually, we achieve sustainable speedup of 15.7× for larger datasets.

TABLE IV
MEMCACHEDB PERFORMANCE COMPARISON

Operation	Moneta-D(ops/s)	Minerva (ops/s)	Speedup
Get	709634	5290015	7.45
Put	415110	4087889	9.85
Workload-A	524842	4686839	8.93
Workload-B	684888	5232544	7.64

C. Minerva based MemcacheDB

To understand the impact on large scale enterprise applications (e.g., Amazon [11], LinkedIn [1] and Facebook [20]) we integrated a Minerva based hash table into MemcacheDB [8], a persistent version of memcached [20], the popular distributed key-value store. By default, MemcacheDB uses BerkeleyDB [21] for reliable persistent storage. MemcacheDB uses a client-server architecture, and, for this experiment, we run it on a single computer acting as both clients (using 16 threads configuration) and server. The key-value data is stored into the SSD. The host sends *get* and *put* requests to the storage processor using commands, and the storage processor performs those operations inside SSD and returns result of execution. We implement *get* and *put* operations in the storage processor and resolve collision using chaining. During *get*, the storage processor applies a hash function on key to get the bucket index. Then it traverses the chain of that bucket in the storage. If the key matches, it returns the corresponding KV data otherwise returns NULL. During a *put*, it applies the hash function to get the bucket index and then inserts the KV pair to the corresponding bucket in the storage. They are implemented as a custom kernel similar to microbenchmarks.

We use YCSB [10] to generate two types of key-value workloads with default configurations of 20-byte keys and 1000-byte values: A) update heavy (50% put / 50% get) and B) read heavy (5% put / 95% get). Table IV shows the throughput of individual operations and workloads performance of Minerva and Moneta-D. Minerva achieves more than 5 M get ops/sec and 4 M set ops/sec which is 7.45× and 9.85× more than Moneta-D. Minerva also outperforms Moneta-D by 8.93× and 7.64× for Workload-A and Workload-B respectively. We improved performance by batching multiple get/set requests in one computation request and performing processing in the storage.

V. RELATED WORK

Intelligent storage Application-level processing on storage originates from research in database machines in the 80's [12]. Those machines did not exist due to the limited disk bandwidth and the complexity of programming special-purpose hardware at that time. With increasing performance of processors and memory in late 90's, the Active Disk [2], [25] project again takes advantage of the processing power on hard disk drives to run application to reduce data traffic between the storage and the host. They proposed a stream-based programming model that allows application code to execute on the disk and identified a set of applications that may benefit from the Active Disk. Intelligent Disk (IDISK) [17] built for decision support database servers was based on similar ideas and provided high-speed serial communication links for communication between IDISKs. However, applications executed on the Active Disk remained I/O-bound due to the disk's poor performance as compared to the DRAM-based main memory. Thus storage vendors have not offered the required software support and interfaces to make the Active Disk practical and widely used.

FPGA-based acceleration Presently, FPGAs have been widely used for various application-specific hardware acceleration. Convey [9] integrates FPGAs along with the general purpose processor which shares the same virtual address space. The IBM Netezza data warehouse appliance [15] uses an FPGA close to the hard drive to reduce data traffic over the network. The emergence of NVMs presents a great opportunity to remove the data access bottleneck in hard drives and match the data processing.

Minerva is based on emerging fast NVM technologies. They are fundamentally different than mechanical disk and flash. The disk technology continues to lag behind the main memory whereas NVMs are narrowing down the performance gap between the storage and the main memory, challenging SSD architecture and I/O interfaces (PCIe, SATA etc.). The previous works on intelligent storage mainly focuses on streaming application to exploit the disk characteristics such as fast sequential access whereas Minerva is not limited to streaming applications only, and supports wide range of I/O and data intensive applications.

VI. Conclusion

We have presented Minerva, a compute capable SSD architecture that provides adequate hardware and software support to efficiently offload application code to the storage array to get advantage of low latency and high bandwidth NVMs and FPGA-based hardware acceleration. Minerva is very suitable for big data applications often exhibit poor temporal and/or spatial locality and perform poorly on the conventional system due to large I/O overhead. We have developed a complete hardware infrastructure that incorporates a storage processor into the storage controller. The storage processor executes application-specific compute kernels, yet co-exists with conventional I/O requests to the storage controller. We have built a high level API for communication between software applications and the compute kernels that abstracts away many details of communication and synchronization. We demonstrate how to map applications to Minerva and evaluate the performance gain. Minerva consistently outperforms the PCM-based SSD that uses the conventional I/O architecture, and achieves speedup of $7.64\times$ to $8.93\times$ on an enterprise workload benchmark.

References

[1] Project Voldemort: A distributed database. Online, Mar. 2012.

[2] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation, 1998.

[3] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A protoype phase change memory storage array. In *Hot Storage: 3rd USENIX Workshop on Hot Topics in Storage and File Systems*, June 2011.

[4] http://www.beecube.com/platform.html.

[5] M. J. Breitwisch. Phase change memory. *Interconnect Technology Conference, 2008. IITC 2008. International*, pages 219–221, June 2008.

[6] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories. In *MICRO 43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, Los Alamitos, CA, USA, December 2010. IEEE Computer Society Press.

[7] A. M. Caulfield, T. I. Mollov, A. De, J. Coburn, R. K. Gupta, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '12, 2012.

[8] S. Chu. Memcachedb. http://memcachedb.org/.

[9] Convey. Convey hc-1: the world's first hybrid-core computer, 2010. http://www.conveycomputer.com/products.html.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[12] D. J. DeWitt and P. B. Hawthorn. A performance evaluation of data base machine architectures (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '1981, pages 199–214. VLDB Endowment, 1981.

[13] G. Dhiman, R. Ayoub, and T. Rosing. Pdram: a hybrid pram and dram main memory system. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 664–469, New York, NY, USA, 2009. ACM.

[14] Fusion-IO. ioxtreme: Extreme performance for extreme users, 2009. http://community.fusionio.com/media/p/463.aspx.

[15] IBM. Ibm netezza data warehouse appliances, 2010. http://www-01.ibm.com/software/data/netezza.

[16] International technology roadmap for semiconductors: Emerging research devices, 2009.

[17] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27:42–52, September 1998.

[18] T. Kolda, D. Brown, J. Corones, J. Critchlow, T. Eliassi-Rad, L. Getoor, B. Hendrickson, V. Kumar, D. Lambert, C. Matarazzo, K. McCurley, M. Merrill, N. Samatova, D. Speck, R. Srikant, J. Thomas, M. Wertheimer, and P. Wong. Data sciences technology for homeland security information management and knowledge discovery. 2005.

[19] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[20] http://memcached.org/.

[21] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db.

[22] http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[23] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[24] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *International Symposium on Computer Architecture*, June 2009.

[25] E. Riedel. Active storage for large-scale data mining and multimedia. pages 62–73, 1998.

[26] Virident. Virident tachion pcie ssd, 2010.